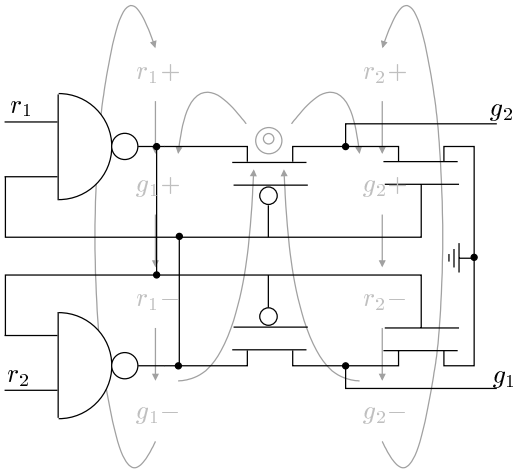
Artur José
Carneiro Pereira

Asynchronous Circuits with Conflicts: a Region-based Synthesis Approach

Síntese de Circuitos Assíncronos com Conflitos: uma Abordagem baseada em Regiões



Artur José
Carneiro Pereira

Asynchronous Circuits with Conflicts: a Region-based Synthesis Approach

Síntese de Circuitos Assíncronos com Conflitos: uma Abordagem baseada em Regiões

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor António Ferrari Almeida, Professor Catedrático da Universidade de Aveiro

O júri

Presidente

Doutor Telmo dos Santos Verdelho
professor catedrático da Universidade de Aveiro

Doutor José Alfredo Ribeiro da Silva Matos
professor catedrático da Faculdade de Engenharia da Universidade de Porto

Doutor António Manuel de Brito Ferrari de Almeida
professor catedrático da Universidade de Aveiro

Doutor António Rui Oliveira e Silva Borges
professor associado da Universidade de Aveiro

Doutor José Carlos Alves Pereira Monteiro
professor auxiliar do Instituto Superior Técnico da Universidade Técnica de Lisboa

Doutor Valery Anatolevitch Sklyarov
professor catedrático visitante da Universidade de Aveiro

Agradecimentos

Quero deixar aqui expresso o meu agradecimento a todos os que, directa ou indirectamente, pessoal ou institucionalmente, contribuiram para que esta tese se tenha concretizado. Há, no entanto, pessoas e intituições que merecem especial referência.

Ao Professor António Ferrari, orientador do meu trabalho de doutoramento, quero agradecer o ter-me "empurrado" para o mundo dos circuitos assíncronos. Quero agradecer-lhe ainda os valiosos comentários e sugestões dados durante a escrita desta dissertação.

Quero manifestar o meu muito especial agradecimento ao Professor António Rui Borges. Ele tem estado sempre presente nos vários passos da minha carreira académica. Neste trabalho de doutoramento, ele foi o colega que me encorajou e sempre acreditou no meu trabalho e foi o crítico que, com a profundidade e rigor que sempre incute ao seu trabalho, muito contribuiu para a clareza que porventura esta tese terá.

O Professor Jordi Cortadella da Universidade Politécnica da Catalunha, merece também o meu agradecimento especial. O tema específico deste trabalho a ele o devo. Foi ele também que, durante várias estadias naquela Universidade, me foi introduzindo em assuntos fundamentais para a prossecução do trabalho e me foi mostrando caminhos a seguir.

Institucionalmente, quero agradecer à Unidade de Investigação da Universidade de Aveiro "IEETA" e ao projecto europeu "Basic Research Working Group 7225 on Asynchronous Circuit Design (ACiD-WG)" por terem suportado os custos relativos aos períodos passados na Universidade Politécnica da Catalunha.

Aos meus colegas de Departamento, docentes e não docentes, incluindo aqueles que neste momento já lá não se encontram, quero deixar o meu agradecimento por me "encorajarem" a abandonar o título de Assistente VCC. Miguel, o título agora é teu; espero que o mantenhas por pouco tempo. Em particular, quero agradecer ao Adrego por, num período de maior aperto, ter assumido um maior esforço de trabalho relacionado com a nossa actividade docente, libertando-me para trabalhar na tese.

Finalmente, quero agradecer à minha família e aos meus amigos o apoio sempre demonstrado. Aos meus filhos, Filipe e Catarina, peço desculpa pelo tempo que lhes roubei para me dedicar a este trabalho.

Resumo

Circuitos assíncronos são uma área de investigação presentemente com um largo número de pessoas envolvidas, quer na indústria quer nos meios académicos. Após um longo período de actividade marginal, tópicos como especificação, análise, síntese ou verificação merecem a atenção da comunidade científica. Uma média anual de publicações superior a 100 durante a última década é disso mesmo uma prova.

A taxionomia habitual de circuitos assíncronos tem por base o modelo de atraso sob o qual se assume aqueles funcionarem correctamente. A classe dos circuitos assíncronos independentes da velocidade (*speed independent asynchronous circuits*), que estão na base do trabalho apresentado nesta tese, assumem um atraso das portas lógicas finito mas sem limite superior conhecido e um atraso dos fios de interconexão nulo ou pelo menos desprezável face ao atraso das portas. A especificação nesta classe é normalmente feita usando dois tipos de grafos: grafos de estados, um formalismo tendo por base os estados do circuito, e grafos de transições de sinais, uma classe de redes de Petri onde se descreve as relações de causalidade e concorrência entre os eventos — transições de sinais — no circuito. Existem disponíveis ferramentas de síntese automática de circuitos assíncronos independentes da velocidade, merecendo *Petrify* a nossa especial referência.

Dois cenários não são contemplados por estas ferramentas, uma vez que infringem uma condição necessária para a existência de uma solução puramente digital independente da velocidade. Um é caracterizado pela existência de não-persistências envolvendo sinais internos ou de saída, situação típica em árbitros e sincronizadores. Uma metodologia de projecto é apresentada que permite a geração de uma solução recorrendo ao uso de ferramentas de síntese para circuitos independentes da velocidade. Um procedimento de transformação toma, à entrada, uma especificação contendo não-persistências e fornece, à saída, um conjunto de componentes especiais, que lidam com as não-persistências, e uma especificação apropriada para alimentar a ferramenta de síntese.

Estabelece-se uma relação entre estados não persistentes e regiões concorrentes, que actuam como secções críticas do sistema. Controlando o acesso a essas regiões, por via da introdução de componentes especiais em hardware, parcialmente analógicos, desempenhando o papel de árbitros, transferem-se os conflitos para os árbitros, ficando o resto do circuito deles isento. Na metodologia proposta, toda a transformação toma a forma de um simples produto de sistemas de transições. Isto resulta da possibilidade de representar os vários passos do procedimento de inserção dos árbitros através de factores multiplicativos. O produto de sistemas de transições goza, se visto em termos de isomorfismo e de grafo alcançável a partir do estado inicial, das propriedades comutativa e associativa, pelo que a ordem de processamento é irrelevante para o resultado final.

O outro cenário corresponde à existência de não-comutatividades entre eventos de entrada. O problema é analisado e diferentes abordagens para o ultrapassar são apresentadas. Uma das abordagens aponta no sentido da transformação das não-comutatividades em não-persistências, aplicando-se de seguida a metodologia desenvolvida para estas. Uma outra abordagem sugere o controlo das não-comutatividades por via da inserção de dispositivos específicos de arbitragem. A análise apresentada deve ser aprofundada por forma a se definir a metodologia mais apropriada para a resolução deste tipo de conflitos.

Abstract

Asynchronous circuits are a subject of research currently with a large number of people involved, both from academy and industry. After a long period of time of marginal activity, topics like specification, analysis, synthesis, verification have deserve attention of the research community. An average of more then 100 papers per year in the last decade in an evidence of that.

The common taxonomy of asynchronous circuits is based on the delay model under which they are assumed to properly operate. The class of speed independent asynchronous circuits, which assumes an unbounded gate delay model, that is, gates have a finite, no upper limited delay while wires interconnecting gates are assumed to have negligible delays, underlies the work presented in this thesis. Specifications are usually described using two types of graph models: state graphs, a state-based formalism, and signal transition graphs, a class of Petri nets. Automatic synthesis tools exist, with *Petrify* deserving our special attention.

Two scenarios in specification are not accepted by these tools, because they infringe a speed independent necessary condition. One is characterized by non-persistences involving non-input signals, which are typical in arbiters and synchronizers. A design methodology is presented that allows the use of existing speed independent tools to derive an implementation for such specifications. A transformation procedure takes a specification with non-persistences at input and delivers both a net list of special components managing the non-persistences and a specification suitable to feed the logic synthesis tool.

Non-persistences are modeled as exclusion relations among regions, which act like critical sections of the system. Introducing special, partial analog components, acting as arbiters, access to these regions are controlled, transferring the conflict points to the arbiters and leaving the remainder of the specification free from conflicts. In the proposed methodology the overall transformation takes the simple form of products of transition systems. In the region-based model used, the several steps for the insertion of an arbiter into the specification can be represented as transition system factors. Thus the product form can be achieved. Up to reachability and isomorphism, the product of transition systems holds the commutative and associative properties. The order of processing of different non-persistences is thus irrelevant to the final result.

The other scenario corresponds to the existence of non-commutativities between input events. The problem is analyzed and different approaches to solve it are discussed. One approach suggests the transformation of the non-commutativities into non-persistences, allowing for the subsequent application of the methodology developed for non-persistences. Another approach suggests the control of non-commutativities by means of the insertion of specific arbitration entities. Non-commutativities must however be further analyzed in order to define and develop a proper methodology to

solve this kind of conflicts.

Aos meus filhos
Filipe e Catarina

À memória de meu pai
Américo Pereira

Contents

List of Figures

Chapter 1

Introduction

Digital circuits are normally seen as arbitrary interconnections of logic blocks, these blocks being simple logic gates or interconnections of smaller logic blocks. The way coordination of activity between logic blocks is carried out leads to the definition of different design styles. There are typically two basic categories: synchronous and asynchronous. The former is characterized by the main assumption that time is discrete. Figure 1.1 shows a generic block diagram of a synchronous circuit. Loops are broken down by memory elements governed by a common periodic signal, the clock. Design problems with feedback paths and hazards — undesired signal transitions — are avoided as long as some time requirements are satisfied. Thus, in order to be properly interpreted, data is required to

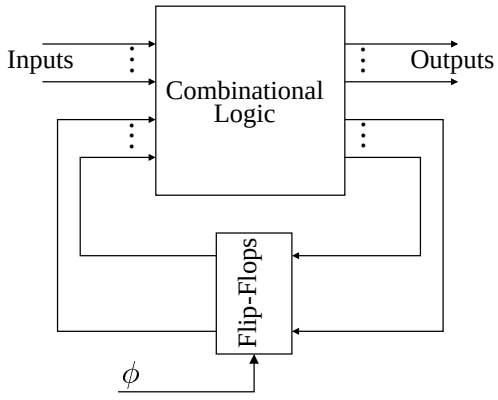


Figure 1.1: Generic synchronous circuit.

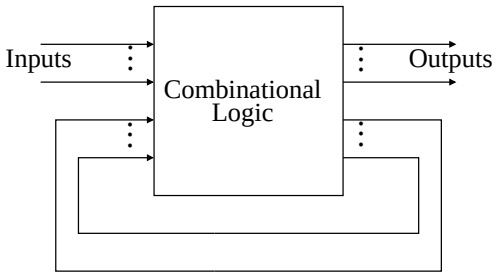


Figure 1.2: Generic asynchronous circuit.

be stable for some minimum amount of time — the setup time — before the clock signal becomes active, and should be held constant for some amount of time — the hold time — while the circuit is active.

Synchronous mode of operation greatly simplifies circuit design. Thus, synchronous design has been widely investigated for a long time. As a result, a large number of techniques and tools for the analysis, synthesis and testing of synchronous digital circuits is available. Synchronous design is the predominant digital circuit design style.

In the asynchronous design style the discrete time assumption disappears. Figure 1.2 shows a generic block diagram of an asynchronous circuit. Feedback lines directly connect the outputs to the inputs, without any clock governed memory element placed in between. Design problems with feedback paths and hazards are now real worries for the circuit designer.

State evolution of a system is no longer commanded by a global clock signal. Coordination of activity between communicating blocks is performed through some kind of *request/acknowledge* protocol. These communications take place only when required by the current computation. Thus, logic blocks remain idle until they are requested to perform some activity. As Brzozowski say in [11] “here lies both the strength and the weakness of asynchronous design: On the one hand, there is a potential for increasing the average speed of the computations and lowering the power consumption. On the other hand, there is a considerable overhead associated with the handshaking protocol”.

Different authors have addressed the advantages and disadvantages of asynchronous circuits relatively to their synchronous counterparts. The following features are often pointed out as benefits of the asynchronous approach [11, 41, 28, 66, 51]:

Average-case performance: When a synchronous circuit is designed we have to guarantee that all the possible computations have been completed before results are latched. Thus, the length of the clock cycle must be long enough to accommodate the worst-case conditions. Many asynchronous circuits work using completion detection mechanisms, thus exhibiting average-case performance. If two functional blocks interact such that one requires the data produced by another, the former can start computation as soon as the latter has completed. This can result, for some cases, in substantial time savings. The ripple carry counter is a good example. Its worst case delay occurs when the carry must be propagated from the least to the most significant bit, and so, is proportional to the number of bits. However, its average-case delay is much smaller.

Low power: In standard synchronous circuits, there is a waste of energy due to useless signal switching. Every clock cycle, the clock drivers and distribution lines are charged and discharged. The clock signal is distributed to all memory elements, even though many portions of the circuit are unused in a given computation. Thus, clock power dissipation represents a significant part of the total power consumption of a circuit. In asynchronous circuits, there is no clock and so, there is no clock power dissipation. When a functional block has nothing to do, all its signals are idled. Thus, for some technologies, like CMOS for instance, its power consumption is quite low, except when it is doing some real work. Asynchronous circuits can require lower power consumption than their synchronous counterparts, and appear well-suited for the design of power efficient systems.

No clock skew: Synchronous circuit designers must devote great care to clock distribution. Clock skew, that is, differences in arrival times of clock events at different parts of the circuit, must be kept within some tight time interval. Often circuit operation must be slowed down in order to accommodate the clock skew. Asynchronous circuits do not suffer from *clock skew* problems, because no globally distributed clock exists.

High modularity: In the synchronous domain no direct benefit is achieved by the replacement of a slow functional unit with a faster one. If the new unit can work at a higher clock frequency, benefits only appear if the clock frequency of the circuit is increased. Often, this implies the redesign of the whole system. If the speed up of the new unit has as a consequence execution

in less clock cycles, substitution of one unit by the other implies redefinition of the synchronization protocol between the unit and the rest of the circuit. Thus, at least in the neighborhood of the unit, redesign is needed. Asynchronous circuits enjoy *higher modularity*. Interaction between units is often based on a self-timed, handshake protocol and so, in general, a circuit accommodates itself nicely to the substitution of one unit by another as long as function and interface remain the same. This easy of composing asynchronous subsystems also allows components from previous designs to be reused.

Easing of global timing issues: In several synchronous circuits, the system clock rate, and thus performance, is determined by the slowest path. The design of all portions of a circuit must be carefully optimized to achieved the highest possible clock rate. This may result in a substantial effort spent in designing rarely used portions of the circuit. Many asynchronous circuits operate at the speed of the path currently in operation. Thus, rarely used portions of the circuit can be left unoptimized without significantly degrading overall performance.

Automatic adaptation to physical properties: Variations in fabrication, temperature and power-supply voltage do change the delay through a circuit. In synchronous design, these factors, namely their worst possible combination, must be considered when determining the clock rate. Asynchronous circuits that sense computation completion adapt themselves to variations in physical properties.

Robust mutual exclusion and external input handling: It is known that elements that guarantee mutual exclusion of independent signals and elements that synchronize external signals with a clock signal are subject to metastability ([13]). A metastable state is a state of unstable equilibrium a circuit can remain in for an unbounded amount of time. Thus, bounded time response, required by synchronous design, cannot be guaranteed. The accommodation of these devices in synchronous circuits is done under the assumption that they will respond in time. Many asynchronous models allow blocks to spend an arbitrarily long time to complete, thus allowing for mutual exclusion elements to leave the metastable state. Also, asynchronous circuits easily accommodate external input signals, since there is no clock with which these signals must be synchronized.

Along with all these potential advantages of asynchronous circuits there are also a number of draw-

backs:

Size: an asynchronous circuit is, in general, larger than its synchronous counterpart carrying out the same function. This may enlarge the computation time of a functional block, narrowing the distance between the average-case performance of an asynchronous circuit and the worst-case performance of a synchronous counterpart.

Power: the potential power savings of an asynchronous circuit can be partially absorbed, or even superseded, by two reasons. In the asynchronous side, there is a power consumption degradation due to a circuit larger size, relatively to a synchronous, equivalent function counterpart. In the synchronous side, techniques have been used to avoid power dissipation in idle blocks. Clock gating and selective powering up/down management are examples of them (see for instance [35]).

Complexity: asynchronous circuits are more difficult to design than synchronous circuits. The designer of a synchronous circuit can simply define the combinational logic necessary to compute given functions, and add some latches to store the results of these computations. By setting an appropriate clock period, worries about hazards — undesired signal transitions — are removed. Contrarily, an asynchronous circuit can be quite sensitive to hazards and so, the designer must guarantee a hazard-free implementation.

Design tools: synchronous designers have at their disposal a large variety of CAD tools to aid them in the various phases of circuit implementation. However, these tools either cannot be directly used for design asynchronous circuits or are hard to adapt.

However the panorama is changing. After a long period of time of null or almost null activity asynchronous circuits started attracting the attention of the scientific community about 16 years ago. Ad Peeters has been collecting asynchronous bibliographic references for the last years [67]. His asynchronous database has, by the time this thesis was written, 1550 entries. The histogram depicted in figure 1.3 was built using that data and shows distribution of the number of publications in the last 50 years. It is clear a growing interest since 1986 with a high activity in the last decade.

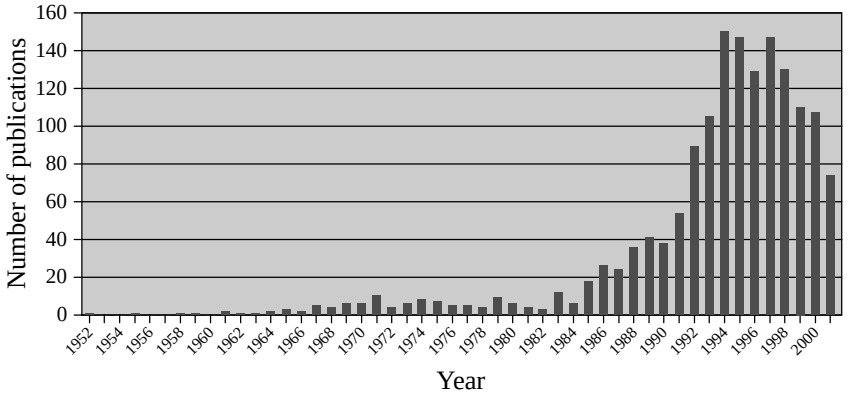


Figure 1.3: Number of publications per year in the field of asynchronous circuits and systems.

1.1 Delay Models

Asynchronous circuits can be represented as interconnections of gates. Gates are logic components carrying out some Boolean function. But gates are physical devices and thus, have inherent delays. If an input of a gate changes at some time, its outputs will respond to this change only some time later. Wires interconnecting the gates also have inherent delays. Therefore, a delay model is fundamental in defining the dynamic behavior of an asynchronous circuit.

Often delays are modeled as single input, single output elements, with some propagation-time magnitude: the output is a delayed version of the input. Gates, on their side, are seen as delay-free components which compute some logic function. Thus an asynchronous circuit appears as an interconnection of components of two types, gates and delays.

Delays are characterized in different ways. From an inertial point of view, a delay can be *ideal* or *inertial* [11, 28]. Every event on the input of an *ideal* delay element is propagated to the output after a certain amount of time. Thus an ideal delay can delay the propagation of a waveform, but does not otherwise alter it. This type of delay is also called *pure* [50] or *perfect* [79]. Figure 1.4.b shows the waveform in figure 1.4.a after passing through an ideal delay element. The ideal delay model is often not realistic, since it does not capture the fact that many physical devices ignore very short pulses. That's what the inertial delay model does. Inertial delays are characterized by a given threshold period. Pulses shorter than this threshold are filtered out by the delay component.

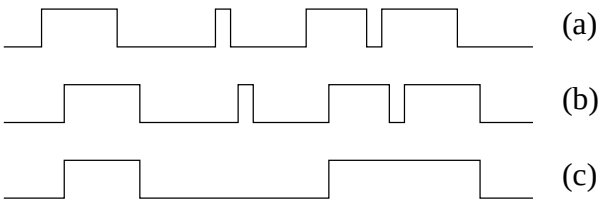
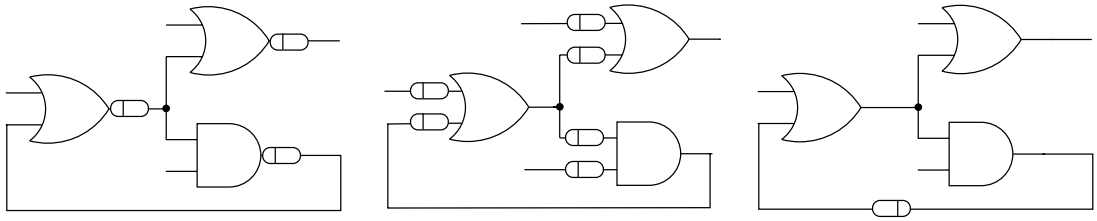


Figure 1.4: Response of an ideal delay element (b) and an inertial delay element (c) to an input waveform (a).

From the magnitude point of view a delay can be fixed, bounded or unbounded. In a *fixed delay model*, the delay has a fixed value. In a *bounded delay model*, lower and upper bounds on the magnitude of delays are known. Brzozowski [11] distinguishes between *bi-bounded* and *up-bounded* delays models. The former assumes delays are bounded both from below and from above by positive constants. The latter assumes the lower bound is zero. The *unbounded delay model* assumes the bound on the magnitude is not known, except that it is positive and finite.

Once the type of delay element has been chosen, a delay model for a given circuit can be defined by the distribution of delay elements in the network of interconnected gates or components. Three alternatives are normally considered for the *circuit delay model*: *gate delay model*, *wire delay model* and *feedback delay model*. All three models are depicted in figure 1.5. In the *gate delay model* a delay is associated with each gate or component in the circuit. Thus, there is exactly one delay element per gate output. This model assumes that wires have a zero delay or that there is no difference among the delays of different branches of the same wire, in which case, the delay of the wire can be added to the delay of the gate. In the *wire delay model* there is exactly one delay element per gate input. Different delay magnitudes among different branches of the same wire can be modeled in this delay model. A circuit is modeled with the *feedback delay model* [50] if the distribution of delay elements in the network is such that every cycle contains at least one delay element and replacing any delay element with a wire produces a circuit in which some cycle contains no delay element.



(a) (b) (c)

Figure 1.5: Circuit delay models: gate delay (a), wire delay (b) and feedback delay (c) models.

1.2 Environment Modes

Every circuit operates in an environment that provides inputs to the circuit and accepts outputs from the circuit. A circuit and its environment form a closed system, called a *complete system* [57]. Thus, given a circuit model, it is important to define the mode of interaction between the circuit and its environment. Three modes of operation are normally considered:

Unrestricted mode: in this mode of operation the environment may change inputs at any time, without paying any attention to the state of the circuit. This is an impracticable model because a circuit might fail to operate correctly if the inputs change too quickly or at a wrong time.

Fundamental mode: in the *fundamental mode* of operation, the environment can only change the input signals when the circuit is stable. A circuit is stable if it is in a state in which its inputs, internal signals, and outputs all have fixed values and have no tendency to change. Put in terms of the circuit delay model, this means all delay elements have their inputs and outputs with the same logical values. In practice this mode of operation is realized estimating the time required for a circuit to stabilize in the worst case, and then making sure that the inputs remain constant for at least that amount of time.

Input-output mode: the *input-output mode* of operation is less restrictive. After changing a circuit input, the environment may change the inputs again only after the circuit has responded by producing an output change, or when no output response is expected [11]. This does not imply that the entire circuit is stable, since some internal signals may still be changing. This mode of operation is used by the more recent asynchronous design techniques.

Considering the number of inputs that can change at a time, a circuit can operate in *single input change mode*, meaning that the environment can only change one circuit input at a given time, or in *multiple input change mode*, meaning that multiple inputs may change in a time interval that is unbounded below. The fundamental mode of operation together with the single input change mode is known as *normal fundamental mode* [50].

1.3 Handshake Signaling

The proper interaction between adjacent modules of an asynchronous circuit is normally based on a *request/acknowledge* protocol. The *request* initiates an action, while the *acknowledgment* is used to signal its completion. For example let there be two modules, a sender and a receiver. The sender requests some action by the receiver, by asserting the request signal. When the receiver is done with the action, it acknowledges its ending by asserting the acknowledgment signal.

Most asynchronous signaling protocols require a strict alternation of request and acknowledge events. There are different ways in which this alternation can be encoded onto specific control wires. The two most common are the *4-phase handshake protocol* and the *2-phase handshake protocol*.

The *4-phase handshake protocol* is represented in figure 1.6. The request and acknowledge events are implemented by the rising transitions of the request and acknowledge signals respectively. There is a release phase — falling transitions of the signals — before the next request/acknowledge can take place. The 4-phase protocol is also referred to as *RZ* (return to zero), *4-cycle* and *level-signaling*. The *2-phase handshake protocol*, also referred to as *NRZ* (non-return to zero), *2-cycle* and *transition-signaling*, is represented in figure 1.7. Now both edges of the request and acknowledge signals are active events.

There are proponents and detractors of both protocols. 2-phase circuits require in general more logic than their 4-phase equivalents. However their proponents argue that they are superior both from a power and a performance standpoint, since every transition represents a meaningful event. On the other side, proponents of 4-phase claim that the time required for the falling transitions of the request and the acknowledge lines does not usually cause a performance degradation, since they happen in parallel with other circuit operations. The 2-phase handshake protocol is used for instance in the asynchronous implementation of the ARM microprocessor [36, 38]. The 4-phase handshake is used

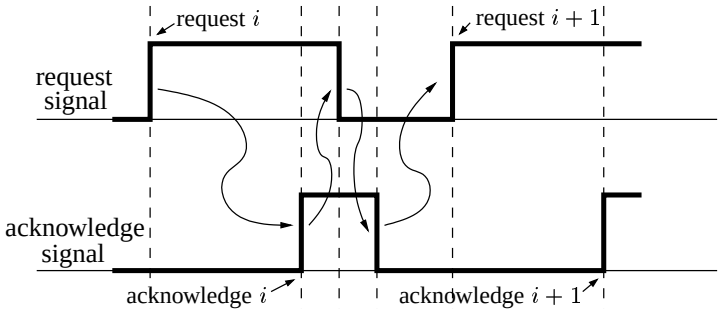


Figure 1.6: The 4-phase handshake protocol.

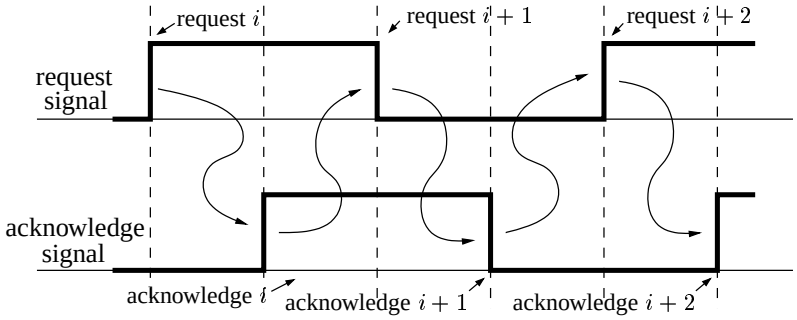


Figure 1.7: The 2-phase handshake protocol.

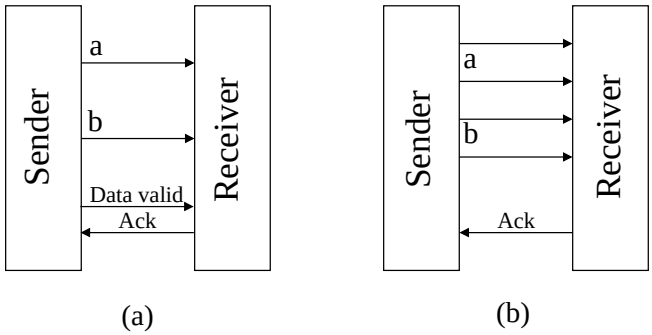


Figure 1.8: Data encoding schemes: (a) bundled data; (b) dual rail.

as the signaling protocol in *handshake circuits* [6], the target asynchronous architecture for Tangram compilations [7].

1.4 Data Encoding

The previous discussion about signaling is only concerned with control wires. There are also choices on how to encode data. The choice for a data encoding scheme is orthogonal to the choice between 2-phase versus 4-phase handshake signaling [8]. The two most common data encodings are *single rail* and *dual rail* [72, 28, 8].

In *single rail* encoding, commonly called *bundled data*, the transmission of n bits of data from one module to another requires n+2 wires, n for the data, 1 for the request or data-valid signal, and 1 for the acknowledge signal. Figure 1.8.a illustrate such scheme. The sender module must assert the data lines and when they are valid must assert the data-valid signal. This relationship of data being valid prior to request assertion must be observed at the receiving side. Otherwise, the receiver could fetch incorrect data values. The acknowledge signal must be asserted by the receiver to acknowledge the end of transaction.

In the *double rail* encoding each bit of data is encoded, together with its own request, onto two wires (see figure 1.8.b). We shall distinguish between *4-cycle* and *2-cycle* dual rail encodings. In the *4-cycle dual rail* encoding the 00, 01, 10, and 11 logical combinations for the wires represent respectively an idle state, a valid 0, a valid 1, and an illegal state. After assertion of a valid data value, the wires must

return to the idle state before the next assertion. The return to the idle state is necessarily commanded by the acknowledge signal coming from the receiver side. In the *2-cycle dual rail* encoding there is neither idle nor illegal states. A valid 0 is represented both by the rising and the falling transition in one of the wires. Similarly, a valid 1 is represented by a transition in the other wire. Again, the sender should not assert a new data value before the previous one has been acknowledged by the receiver side.

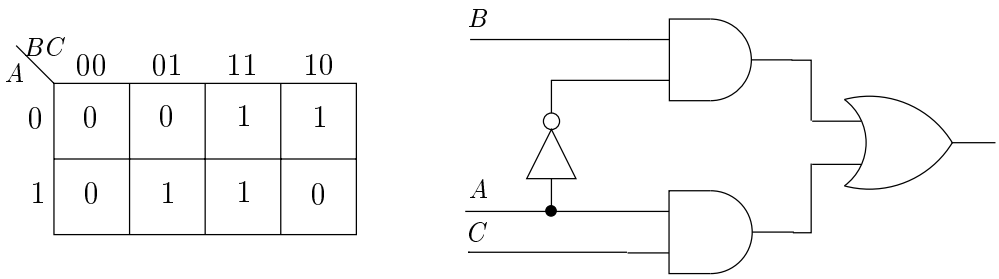
There are advantages and disadvantages for the use of both the bundled data and dual rail encodings. Bundled data encoding is more conservative in terms of wires. For encoding n bits of data n+2 wires are required, while the dual rail encoding needs 2n+1 wires. Dual rail encoding is insensitive to the delays on any wires, while the bundled data encoding does contain an implied timing assumption. Therefore, dual rail is more robust, specially when the timing constraints cannot be guaranteed. Also, the dual rail encoding is less efficient in terms of power consumption. The asynchronous implementation of the ARM microprocessor [36] and the handshake circuits [8] are examples of usage of respectively the bundled data and the dual rail encodings.

1.5 Hazards

Glitches on wires are not usually a problem in synchronous design, as long as transitions stabilize before the next active clock transition. However, since asynchronous circuits have no global clock, a glitch may cause a system to fail. The potential for a glitch in an asynchronous design is called a *hazard* [28, 76, 11].

There are two types of hazards in combinational circuits: *static hazards* and *dynamic hazards*. A *static hazard* causes an output to change transiently when it was supposed to remain stable, due to changes in one or more input signals. Consider the combinational circuit in figure1.9. If the circuit is in state $ABC = 111$ and a change occurs in input A, the output should remain at 1. However, if the delay of the lower AND-gate is smaller than the sum of the delays of the inverter and the upper AND-gate, a glitch $1 \rightarrow 0 \rightarrow 1$ will appear at the output of the OR-gate. More specifically a static hazard is called a *static-1 hazard* if the glitch has the form $1 \rightarrow 0 \rightarrow 1$, and *static-0 hazard* if the glitch has the form $0 \rightarrow 1 \rightarrow 0$.

A *dynamic hazard* causes an output to change three or more times when it was supposed to change



(a) (b)

Figure 1.9: A circuit with a static-1 hazard.

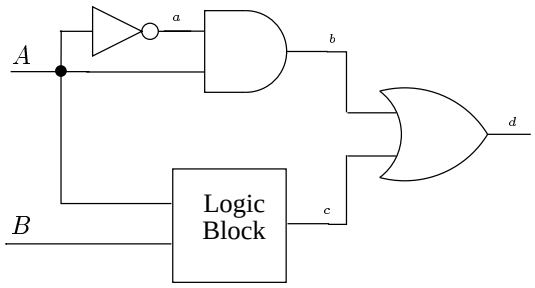


Figure 1.10: A circuit with a dynamic hazard.

only once due to changes in one or more input signals. Consider the circuit in figure 1.10, and assume it is in state $AB = 01$ and input A changes from 0 to 1. Consider that due to the gate delays, variables a, b, c and d may change in the following order: b, d, a, b, d, c and d (we are assuming that c changes as a consequence of the change in A). Thus, the output of the OR-gate, which should change from 0 to 1, actually manifests the following sequence of transitions: $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$.

Additionally to combinational hazards, there are hazards due to the sequential nature of asynchronous circuits. There are two types: *critical races* and *essential hazards*. When a sequential asynchronous circuit changes from one state to another, several state bits may change. Eventually, the circuit may stabilize incorrectly in a transient state. In such cases a *critical race* occurs. A proper state encoding technique can be used to avoid race conditions. The one-hot state encoding technique is one example. The basic idea behind this encoding method, proposed by Hollaar ([43]), is to assign one bit of

memory to each state.

Essential hazards [28, 76] arise if a sequential circuit has not fully absorbed an input change at the time the next state begins to change. That is, the circuit sees the new state before the combinational part has stabilized from the input change. Essential hazards are avoided by adding delays to the feedback path.

1.6 Design Methodologies

Using the delay models described in section 1.1 a circuit model can be defined. Then design can take place. For this a formalism to describe (specify) the behavior of a circuit and a method to derive (synthesize) an implementation from the specification are necessary. Eventually, the method can be partially, or even totally, automated and a synthesis tool can be developed. Some design methodologies can support more than one delay model. For instance, data and control paths of a circuit can be modeled using different delay models.

1.6.1 Huffman Circuits

For historical reasons we shall start referring to the *Huffman model*, so called because of the work of Huffman (see [44, 45] and [76]). The model decomposes a sequential circuit into a combinational part and feedback loops. These feedback loops store the states of the circuit, so no flip-flops or latches are used. Eventually, the feedback loops may have added delays. Figure 1.11 shows the general block diagram of a Huffman sequential circuit. The combinational part is assumed to work under a bounded, inertial, wire delay model. Feedback lines, which correspond to memory elements in synchronous design, are assumed to have unbounded, inertial delay.

Specification is done at state level, by means of a *flow table* or *state table*, a sort of truth table where next state and output signal values are defined as a function of present state and input signal values. As shown in figure 1.12, the flow table has a row for each internal state, and a column for each combination of inputs. Entries in table cells indicate next state entered and output generated when column's input combination is seen while in row's state. Present state and next state are represented

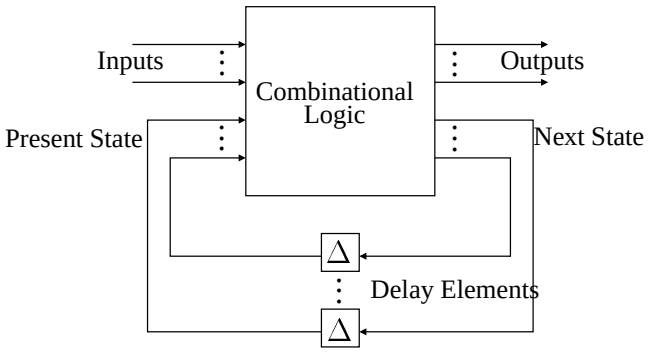


Figure 1.11: Block diagram of a Huffman circuit.

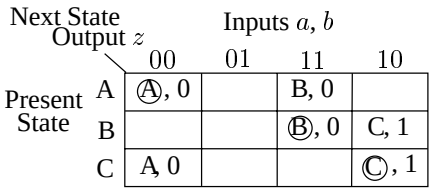


Figure 1.12: A flow table.

symbolically. *Stable states*, states where the next state is identical to current state, are represented encircled.

The synthesis process, similar to the synthesis of finite state machines in synchronous systems, unfolds into 3 steps: flow table minimization (state reduction), state encoding and hazard-free implementation. Some aspects must be considered during the synthesis process. Since there is no clock to synchronize input changes, the system must deal correctly with intermediate states caused by multiple input changes. In figure 1.12, a change from present state A to next state B occurs after a change in the inputs from 00 to 11. But the change in the inputs is not direct, it will briefly pass through 01 or 10. Entries for both inputs 01 and 10 must be added to the row, which keep the system in state A.

Huffman circuits are usually coupled with the fundamental mode of operation. Also, usually, the single input change mode is imposed. For this the hazard-free implementation of combinational logic is quite simple. All static and dynamic hazards can be eliminated by adding certain products to a sum-of-product realization of the circuit [76]. Unfortunately, this method cannot guarantee correct

operation under the multiple input change mode.

The single input change mode together with the fundamental mode of operation forces some requirement on the feedback lines and on the state encoding schema. It must be ensured that the combinational logic has settled in response to an input change before a state variable changes. This is done by placing delay elements on the feedback lines with a proper propagation-time magnitude. The same restriction imposes that only one state variable can change at a time. A state encoding schema in which a single state bit changes in each state transition meets this requirement. However, these encodings sometimes require multiple representations of the same state, which complicates the combinational logic [76]. An alternative is to use the *one-hot state encoding*, in which each state is encoded by a different state variable. This schema requires two state variable changes per state transition. A state transition is accomplished by first setting the state variable of the next state and then resetting the state variable of the current state. The final requirement is that the next input change can only occur after the entire circuit has reached a stable state.

1.6.2 Burst-Mode Circuits

A design methodology, called *burst-mode circuits*, relax the single input change mode of operation of Huffman circuits, and moves closer to synchronous design style. Behavior, in this design methodology, is specified using a standard state machine, where each arc is labeled by a non-empty set of input events — an input burst — and by a set of output events — an output burst. Figure 1.13 shows the burst-mode specification for a controller developed at HP Labs [26]. Each arc is labeled with an input burst and an output burst, with a slash in between.

When the circuit is in a given state, one of the input bursts leaving that state can occur. Changes of inputs in the burst can occur in any order and the circuit can only react when the entire burst has occurred. In order to unambiguously determine when an entire burst has occurred, no input burst can be a subset of any other input burst leaving the same state.

Once an entire burst has occurred, the circuit activates the associated output burst and enters the associated next state. New input changes are allowed only after the circuit has stabilized in reaction to the previous input burst. Thus, there is a kind of fundamental mode of operation, but only between signal changes in different input bursts.

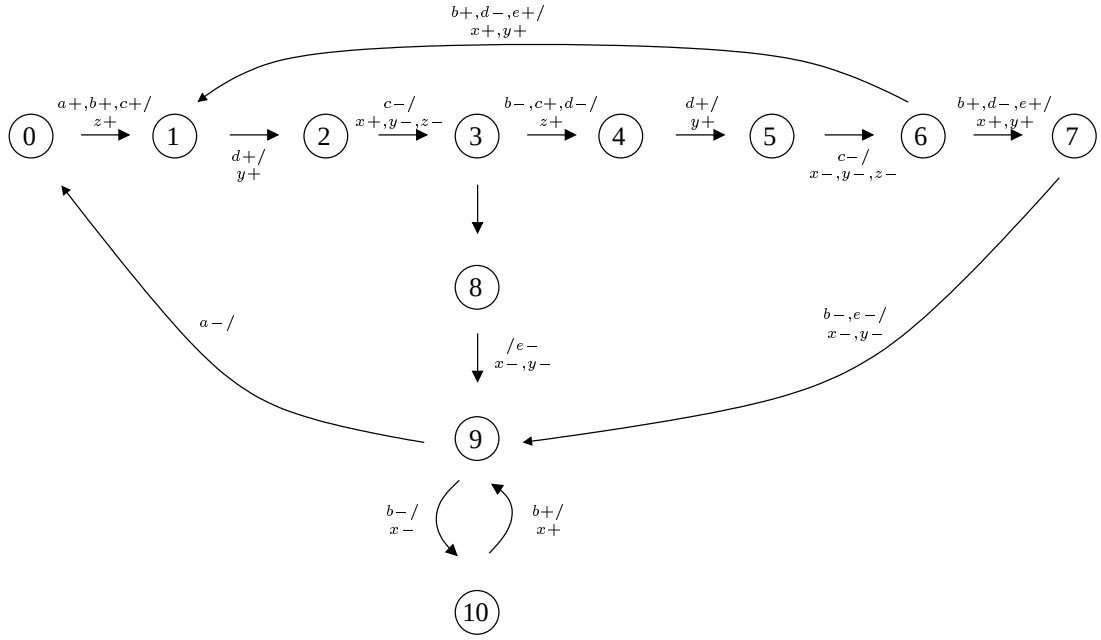


Figure 1.13: Burst-mode specification, borrowed from [28], of a controller of the Post Office routing chip [27].

Burst-mode specifications are often synthesized into a target architecture, called *locally-clocked state machine* [63, 64]. Figure 1.14 shows the general circuit schematic for a locally-clocked implementation. This circuit is decomposed into pieces:

- Clock generator logic, which produces a clock pulse whenever the state signals must change.
- A set of two-phase clocked memory elements.
- Combinational logic implementing the next state functions.
- Combinational logic implementing the output functions.

A clock signal is generated locally in each state machine, and is independent of the local clock of any other module. Suppose that in response to an input burst one or more state bits must change. In that case the state machine reacts as follows. First, the combinational logic for the output and the next state change in response to the inputs. At the same time the local clock logic is getting ready to start the clock pulse. However, delays are added to the clock line in order to guarantee proper

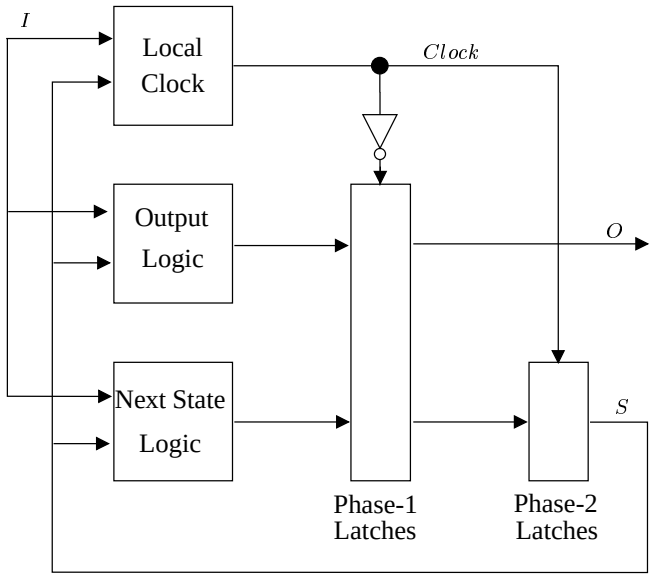


Figure 1.14: General schematic for a locally-clocked circuit implementation.

propagation through the phase-1 latches before the clock fires. Once the clock fires, the phase-1 latches are disabled, while the phase-2 latches become enabled. Next state can now flow back through the feedback lines into the next state, output, and local clock logic blocks. Since the phase-1 latches are disabled no hazards are produced. The local clock is then reset, phase-2 latches are disabled, and phase-1 latches are enabled again. The reaction cycle of the state machine is completed, and it is ready for a new input burst.

An alternative implementation style for burst-mode specifications was proposed by Yun and Dill [85, 84]. They are called *3D asynchronous state machines* and are implemented by techniques similar to those used for Huffman circuits, with no local clock or latches.

1.6.3 Micropipelines

Micropipelines are a powerful design methodology developed after initial work by Ivan Sutherland ([75]), primarily intended as an asynchronous alternative to synchronous elastic pipelines, i.e., pipelines in which the amount of data contained can vary. They are based on a specific structure in which data modules are encased into control chains. Data modules compute some useful work. Con-

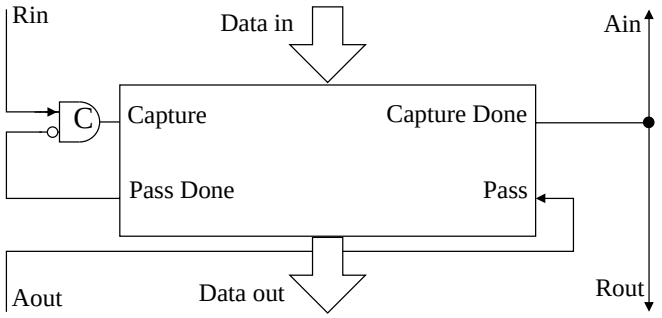


Figure 1.15: A micropipeline event controlled latch.

trol path manages communication of data between adjacent stages. It adopts a *bundled data* protocol for the data path and a *two-phase handshake* protocol for the control path.

An essential building block for a micropipeline circuit is the *event controlled latch* [36] shown in figure 1.15. The gate labeled C in the figure is a Muller C-gate. It performs the rendezvous function for events: it waits until it has received an event on both of its inputs before issuing an event on its output.

Assume the latch begins in a transparent state. The C-gate has one of its inputs circled. It means that we assume that initially an event has already occurred on that input. The occurrence of an event on the other input, the request line R_{in}, indicates that data is valid on the "data in" lines. This event passes through the C-gate to the "capture" input, causing the data to be latched. When the latch has captured the data it issues an event on the "capture done" line, which is used for two purposes. On one hand, it is used as an acknowledge event, indicating that the data has been captured and so, the "data in" lines are available for new input data. On the other hand, it is used as a request out event, stating that data is valid on the "data out" lines. This data is held stable until an acknowledge event is received on the A_{out} line. At that time the latch is put back into transparent (pass) mode and the event is propagated to the C-gate input. The latch is now ready for a new cycle. Note that by the time the "pass done" event occurs a request can be pending on the other input of the gate. Actually, the C-gate ensures the correct operation of the latch whatever the relative timings on the R_{in} and A_{out} events.

Through cascading event controlled latches and logic blocks, as is shown in figure 1.16, a complete computation pipeline can be built. Ignoring first the logic blocks and the explicit delay elements, one

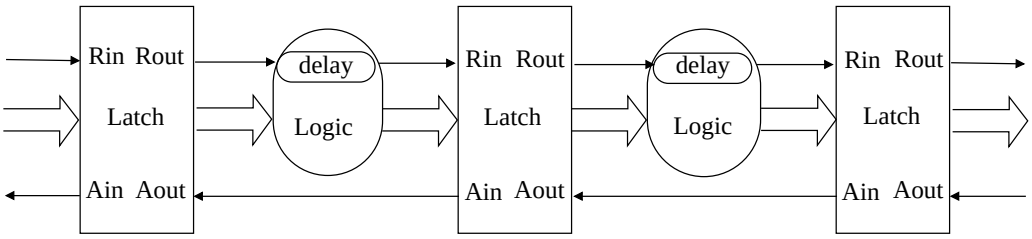


Figure 1.16: A micropipeline with computation.

gets a simple data FIFO. A data value can be placed on the left most latch by signaling an event on the R_{in} input. That data then moves through the intermediate latches to the right most one at a speed determined only by properties of the technology from which the circuit is constructed. As soon as a latch captures a data value, it sends an acknowledge back signaling that a new data value can be processed.

At each stage, computation is performed by the logic blocks. These blocks introduce some delay on the data moving through them. Thus, a delay element must be added into the request line in order to ensure that the bundling constraint is met at the request input event of the next latch. The propagation magnitude of the delay element must be as large as the worst-case delay of the logic block.

A major benefit of a micropipeline structure is that hazards in each logic block are absorbed by the latch put in front of it. Thus, any logic structure, including those used in synchronous design, can be used to implement the logic blocks.

The AMULET group at the University of Manchester has put a large effort into addressing asynchronous design using micropipelines. They have developed and fabricate the AMULET1 [80], an asynchronous implementation of the ARM [37] 32-bit RISC microprocessor. More recently they have delivered AMULET2e [40, 39], an asynchronous embedded controller which incorporates AMULET2, an enhanced version of AMULET1, and several other blocks.

1.6.4 Delay-Insensitive Circuits

Delay-insensitive circuits are a quite interesting theoretical class of circuits. They assume an unbounded wire delay model: more robustness is not possible, since it can accommodate any magnitude

and distribution of delays. However the delay-insensitive assumption has a great impact on circuit implementation. Consider two communicating modules, a sender and a receiver. There is no guarantee that an event produced by the sender has been properly received by the receiver, no matter how long one waits. Thus, the receiver must necessarily inform the sender, by an acknowledge event. In turn, the sender must wait until it receives the acknowledge before sending a new event. Both the 2-phase and 4-phase signaling protocols described in section 1.3 can be used. The transmission of data also suffers from the same limitation. There is no guarantee that a wire will reach its proper value at any specific time. Thus, the bundling data encoding method cannot be used, and the dual rail encoding must be used instead.

In its pure assertion, delay-insensitive circuits are of little practical interest, due to severe limitations ([29, 55]). To make delay-insensitive circuits suitable for general computations, we need a set of basic blocks that both work properly under the delay-insensitive assumption and provide sufficient functionality to implement practical circuits. It is known that standard gates are not suitable [11, 55]. Thus, incursions with delay-insensitivity are done by relaxing the model somehow.

Martin [54] has assumed some *forks* are isochronic. A *fork* is a wire with two or more branches. A fork is *isochronic* if the delay differences between different branches are negligible. These class of circuits were called *quasi-delay insensitive*. The list of forks that must be isochronic is part of the synthesis output data.

A different relaxing approach assumes a circuit as a delay-insensitive interconnection of modules, while the modules themselves are designed using a more rigid model. This is in general an acceptable compromise. A basic module usually involves a relatively small area on a chip and its delays can be quite well controlled. The works by Ebergen [33, 34], Kees van Berkel [6] and Brunvand [10], among others, fall in this category.

Specification is usually described as a program in a high-level language. Such languages are typically similar to Hoare's communicating sequential processes [42] and Dijkstra's guarded commands [30]. The program is then transformed, through a series of steps, into a low-level program which maps directly to a circuit. Transformations are carried out using algebraic manipulation or compiler techniques. In order to illustrate the method, next we are doing a brief incursion in one of these methodologies, the Ebergen's trace theory.

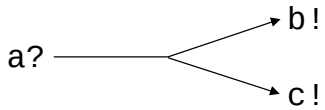
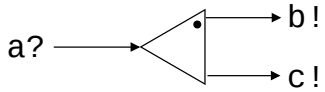
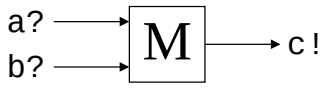
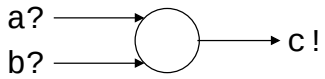
WIRE		`pref*[a?;b!]`
FORK		`pref*[a?;(b!||c!)]`
TOGGLE		`pref*[a?;b!;a?;c!]`
MERGE		`pref*[(a?|b?);c!]`
JOIN		`pref*[(a?||b?);c!]`

Figure 1.17: Some basic delay-insensitive modules and their corresponding commands in Ebergen's trace theory.

Ebergen's trace theory

Ebergen introduced a design methodology for delay-insensitive circuits based on commands. A *command* is a program construction, similar to a regular expression, used to describe a circuit functionality. Several operations are available to construct complex commands from simpler ones. These operation are: *concatenation*, *union*, *repetition*, *weave* and *prefix-closure*.

Figure 1.17 shows a number of delay-insensitive basic modules or components, and their corresponding description command. The *wire* is a component with one input, $a?$, and one output, $b!$. The question-mark ("?") indicates an input to the wire, while the exclamation-mark ("!") indicates an output of the wire. Under the delay-insensitive assumption, the correct operation of a wire imposes that the input and the output events must strictly alternate, that is, once a change on $a?$ has occurred, no more changes on $a?$ are permitted until a change on $b!$ has occurred. Otherwise, two successive

changes on $a?$ may result in a glitch on $b!$. The behavior of a wire is described by the command `pref*[a?;b!]`. The semi-colon (";") denotes *concatenation*: the input event $a?$ must be followed by the output event $b!$. The asterisk ("*") denotes *repetition*: events $a?$ and $b!$ may alternate any number of times. The *prefix* operation means that any prefix of a permitted behavior is also permitted.

On delay-insensitive design different branches of the same wire must be considered separately. The *fork* is the basic component representing a pair of branches in a wire. Its behavior is described by the command `pref*[a?;(b!‖c!)]`. The input event $a?$ must be followed by both output events, $b!$ and $c!$. The order of occurrence of the two output events is irrelevant. This parallel composition of events is described in the command by the parallel bar ("‖") operator, called the *weave* operator.

The *toggle* component has one input, $a?$, and two outputs, $b!$ and $c!$. Each input event must be followed by exactly one output event. But output events must alternate (toggle), that is, if an input $a?$ results in output event $b!$ ($c!$), the next input event $a?$ must result in output event $c!$ ($b!$). The first input event $a?$ must result in the output event marked with a black dot. The overall behavior of the toggle component is described by the command `pref*[a?;b!;a?;c!]`.

The *merge* component has two inputs, $a?$ and $b?$, and one output, $c!$. It implements an exclusive choice between the two inputs. It waits for exactly one input event, and once it has occurred responds with the output event. The choice is represented in a command by the choice bar ("|") operator, which is called the union operator. The final command for the merge component results in `pref*[(a?|b?);c!]`.

The last basic module is the *join*. It has two inputs, $a?$ and $b?$, and one output, $c!$, and implements the rendezvous of events. The component waits for events on both inputs and once they have occurred produces an event on the output. The input events may occur in any order, but different occurrences of the same input must be interleaved by the output event. The total behavior of the join component can be given by the command `pref*[(a?‖b?);c!]`. Above, we have said the weave operation corresponds to the parallel composition. However, it actually corresponds to a parallel composition with synchronization on common symbols. That fact allows the command `pref*[(a?;c!)‖(b?;c!)]` to be an alternative description for the join component.

A command in Ebergen's trace theory can also be used to specify a complex circuit. For instance, the command

```
pref*[a?;b!;a?;b!;a?;c!]
```

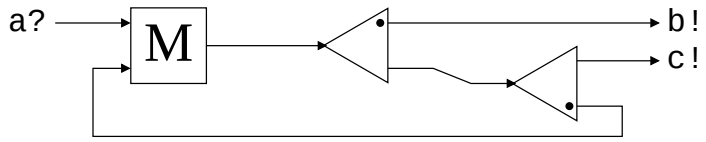


Figure 1.18: Delay-insensitive implementation of a module-3 counter using Ebergen's methodology.

represents the behavior of a module-3 counter. The input $a?$ must occur three times before the output event $c!$ occurs. Because of the delay-insensitive assumption, each input event must be acknowledged by an output event. Thus, the need for the output $b!$.

A command can be decomposed, through a series of steps, into an equivalent network of basic components. In Ebergen's methodology this is done applying algebraic manipulation. The module-3 counter can be decomposed into a network of two toggles and one merge components, as is shown in figure 1.18.

1.6.5 Speed-Independent Circuits

Speed independent circuits use the unbounded gate delay model: gates are assumed to have a finite, unbounded delay, while wires are assumed to have negligible delay, that is, much lower than the smallest gate delay. In practice, the restriction on wire delays can be relaxed by the isochronic fork assumption. As mentioned before, a fork is isochronic if the difference in delays between branches is negligible. Delays on wires with a fanout of 1 can simply be considered as part of the gate delay. If wires have branches but delay differences between them are negligible, the same consideration applies. This is illustrated in figure 1.19. On the left side, it is depicted a circuit with a 2-output fork. If ϵ is negligible, the fork can be assumed to be isochronic and the circuit on the right side is its equivalent speed-independent form.

Speed independent circuits are attractive for a variety of reasons. Among others:

- Although more restrictive than delay insensitive circuits, they are still quite robust. Variations in gate delays are well tolerated, not introducing unexpected behaviors. For instance, this makes migration to a faster technology easy. Only performance is eventually altered.

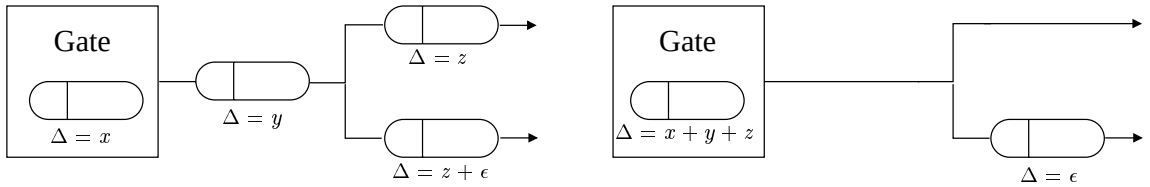


Figure 1.19: Two equivalent 2-input forks: (a) wire delays are explicitly expressed; (b) common part of wire delays were transferred to the gate delay. If ϵ is negligible, that is, it is almost zero when compared with the gate delays, the fork in (a) is isochronic, while the one in (b) has a speed-independent form.

- In several circuit implementation technologies the speed-independent delay model is acceptable, making it of practical interest.
- Standard library components can eventually be used. Complex specifications can be built up from combinational gates, like AND- and OR-gates, and asynchronous memory elements, like SR-latches and C-elements.
- The design of complex systems can be decomposed into sets of communicating modules. As long as module interfaces are respected each module can be synthesized independently from the others, thus giving a high modularity. Additionally, module redesign is possible without the need to global redesign.

Circuit specification

Since the pioneering work by Muller [60] several incursions into the design of speed independent circuits have been taken. Specification of circuit behavior is normally given using state- and event-based graph models. In state-based models, system behavior is described by means of an asynchronous state machine. States of the system, as expected, are represented by vertices. Signal transitions, which cause the system to change from one state to another, are represented by labeled arcs. However, each label is assumed to represent a single signal transition. That is, the dynamics of the model assumes that signal transitions occur only one at a time, and thus, concurrency is not explicitly represented. The state machine changes state whenever a signal transition occurs. Figure 1.20.a shows an asyn-

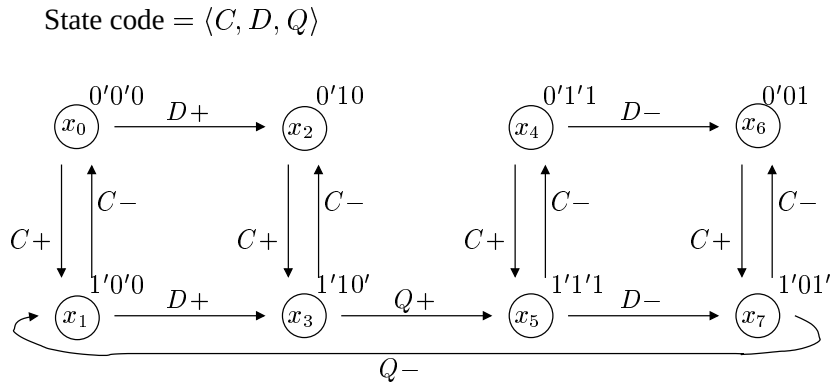


Figure 1.20: A state graph description of a D-latch.

chronous state machine describing the behavior of a D-latch. Note that the state machine specification describes not only the behavior of the circuit, but also the expected behavior of the environment. For instance, in state x_3 it is assumed the environment cannot fire event $D-$; it must wait until $Q+$ has fired before firing $D-$.

A Petri net [61, 69] is an event-based graph formalism that can explicitly represent causality, concurrency and choice. A Petri net is depicted in figure 1.21.a. It contains two kinds of vertices: *places* and *transitions*. Transitions are drawn as bars and are bounded to events. Places are drawn as circles and represent the pre- and post-conditions for the occurrence of the events. Each place can hold zero or more *tokens*, which are drawn as black dots. An assignment of tokens to the various places of a net is called a *marking*, and represents a state of the system. A transition is enabled to *fire* if all its predecessor places are marked, that is, have at least one token. An enabled transition can fire, and when it does — its bound event occurs —, there is a movement of tokens from the predecessor places to the successor ones. Then the current marking changes, that is, the system changes state.

A Petri net, where transitions are interpreted as signal transitions, allows a designer to capture the behavior of an asynchronous circuit in a manner quite similar to timing diagrams. In this way, the circuit is viewed not as a state machine, but rather as a partially-ordered sequence of events. The Petri net in figure 1.21.a actually represents the behavior of the D-latch, whose state graph description is given in figure 1.20.

Currently, the widest-used event-based formalism is the *signal transition graph (STG)*. It corresponds

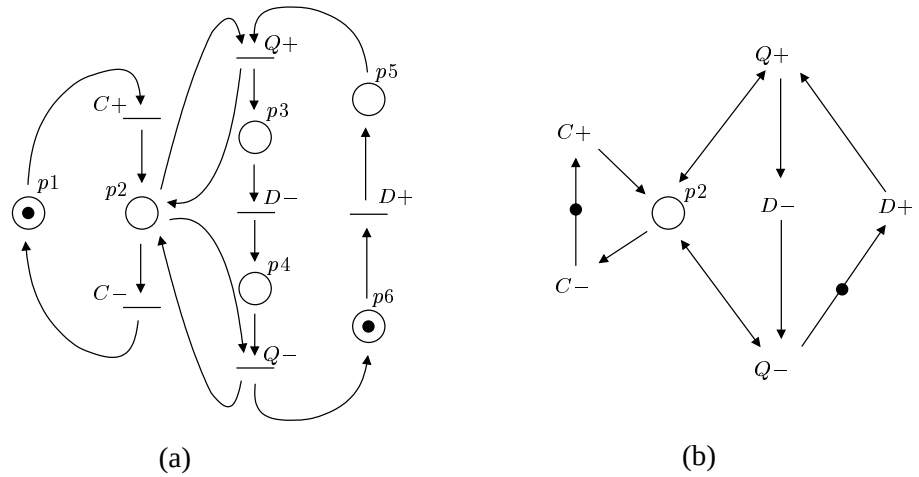


Figure 1.21: (a) A Petri net describing the behavior of a D-latch. (b) An equivalent STG description.

to the previous Petri net with a slightly simplified pictorial representation. Bars in transitions have been dropped and thus, only the label (signal transition) is used. Places with single input arc and single output arc are omitted and a single arc connects preceding and succeeding transitions. Tokens in these "hidden" places are represented as dots on the arc. Figure 1.21.b shows an STG equivalent to the Petri net at its left side. STG were introduced by Chu *et al.* [14, 15], although with much more severe restrictions than the ones accepted nowadays. Two similar models were independently introduced, namely the *signal graphs* [70, 79] and the *change diagrams* [46, 47].

Playing the token game on a Petri net or STG one can generate a directed graph, where vertices correspond to markings and arcs correspond to transitions between markings. This graph is called the *reachability graph* of the Petri net. The asynchronous state machine in figure 1.20 is the reachability graph of the Petri net (STG) in figure 1.21.

Implementability properties

Each state of an asynchronous state machine, like the one in figure 1.20, and each marking of a signal transition graph can be associated with a binary state vector, in which each bit represents the state, either 0 or 1, of a circuit signal. Input, output and internal signals are represented in the state vector. Bits for excited signals in each state are also marked with a prime. States in figure 1.20 are labeled

with the corresponding binary state vector. An asynchronous state machine with binary encoding is called a *state graph* [16]. However, often the term state graph is also used when the state labels are not present.

A state graph or a signal transition graph is *consistent* [48] if rising and falling transitions alternate for each signal, that is, there is never an attempt to raise a signal already high or lower a signal already low. Consistency is a necessary condition for the realizability of a specification as a circuit.

But it is not sufficient. Chu [16, 58] has formulated a necessary and sufficient condition for the existence of a circuit implementation of a consistent specification. It is called the *complete state coding* (CSC) property and determines that all markings of a STG (states of a SG) with the same binary code must have the same set of enabled non-input signal transitions.

Finally, another property must be hold by a specification if it is to be implemented as a speed-independent circuit [48]. It states that:

1. non-input signal transitions can not be disabled by any other signal transition;
2. input signal transitions can not be disabled by any non-input signal transition.

The former condition ensures that no glitches can appear at the gate outputs. It also avoids meta-stability or oscillation that can appear due to mutual output disabling. The latter ensures no hazards can occur at inputs of the circuit. Disabling between input signal transitions is assumed to be correctly managed by the environment.

The state graph description of the D-latch (figure 1.20) is consistent, holds the CSC property, but it is not speed-independent. In state x_3, output event $Q+$ is disabled by the firing of input event $C-$. Output event $Q-$ in state x_7 is disabled by the same event.

Logic synthesis

Logic synthesis of speed-independent circuits can take place from both state graph and STG specifications. Starting from a state graph specification, logic synthesis unfolds into the following steps [48]:

1. encoding the SG in such a way that the complete state coding property holds;

2. deriving the logic functions for output and internal signals;

3. mapping the functions onto a netlist of gates.

For each output or internal signal v, the next-state function maps state vector codes into the set $\{0, 1, -\}$, which respectively represent logical values 0, 1 and don't care. The on-set is composed of all state vectors with signal bit stable at 1 or excited at 0. Similarly, the off-set is composed of all state vectors with signal bit stable at 0 or excited at 1. Finally, the don't care-set is composed of all state vectors of states not reachable from the initial state.

It is quite clear now the necessity of the complete state coding property. If it does not hold, the same state vector code goes to both the on- and off-sets. Thus, there is a conflict in the definition of the function. This problem can eventually be solved inserting new internal signals, which eliminates the encoding conflicts. The insertion of the new signals must be done in such a way that the resulting state graph keeps consistency and speed-independence.

Once a state graph specification holding the CSC property is found, next-state functions for the output and internal signals can be derived. Boolean minimization can then be performed to obtain the logic equations implementing the behavior of the signals. Efficient use of don't care conditions plays a central role in this step.

Finally, the previous equations must be mapped to components of a hazard-free library of gates. Traditional logic synthesis of speed independent circuits assumes the existence of non-standard implementation libraries, such as arbitrary complex gates [15] or arbitrary fan-in AND-gates [4, 49]. On the other hand, standard logic decomposition followed by technology mapping can not be applied because hazards may be introduced. Some recent work has addressed the logic decomposition and technology mapping issue. See for instance [74], [5], [71], [12] and [48]. The approach referred in [48] aims at solving the problem of speed-independence-preserving decomposition of large logic gates into smaller ones, namely two-input NAND or NOR gates. The method has been embedded into the overall synthesis procedure of Petrify [18], a publicly available software tool.

When one starts from an STG specification, logic synthesis can be addressed by two different approaches. In one of them the STG is traversed and its reachability graph is determined. Then logic synthesis proceeds as stated before.

As concurrency grows a reachability (state) graph can be exponentially larger than the corresponding STG specification. To overcome this state explosion drawback some approaches address synthesis directly from the STG specification, without explicitly or implicitly deriving the state graph. However they impose more restrictions to the STG specification than SG approaches do. For additional information on this topic see [52], [65] and [83].

1.7 Arbitration

Arbitration and the need for mutual exclusion is a topic the designer of a concurrent system must be aware of. An asynchronous circuit is by nature a concurrent system and thus, arbitration phenomena can be present and must be considered in the design. Let us consider, for instance, the behavior of the D-latch, depicted in figure 1.20. In real circuits signal transitions are not instantaneous. Thus, in state x_3, if the environment decides to fire transition $C-$ during the occurrence of event $Q+$, there is a "fight" between keeping signal Q low and changing it to high. In order the circuit works properly, events $C-$ and $Q+$ must occur in mutual exclusion, which in a typical pure digital implementation of the D-latch corresponds to impose some time constraints to the specification.[1] This violates the speed-independent assumption.

An *arbiter* is a device that grants the access to a common resource to exactly one of a number of processes requesting it. Circuits implementing arbiters can also run into anomalous behavior if requests from different processes arrive within a time interval shorter than the delay of a transition process completion in the arbiters [13, 79].

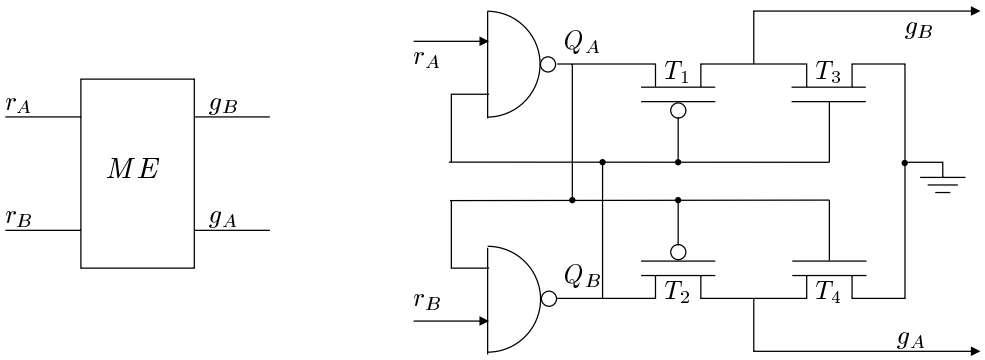
There are two main types of anomalous behavior to be considered: meta-stability and oscillatory anomaly. Meta-stability is characterized by a state of unstable equilibrium, in which the signal outputs coincide on a level representing neither a logical 0, nor a logical 1. The oscillatory anomaly is characterized by an oscillation of the outputs between the two logical values.

Once a circuit enters a meta-stable state, it can remain on it for an unbounded amount of time, before

[1] The typical implementation of the D-latch corresponds to the logic function

$$Q = D.C + (D + \overline{C}).Q$$

which only works correctly assuming that transitions of D occur far enough from the falling edge of C, the well-known setup and hold times.



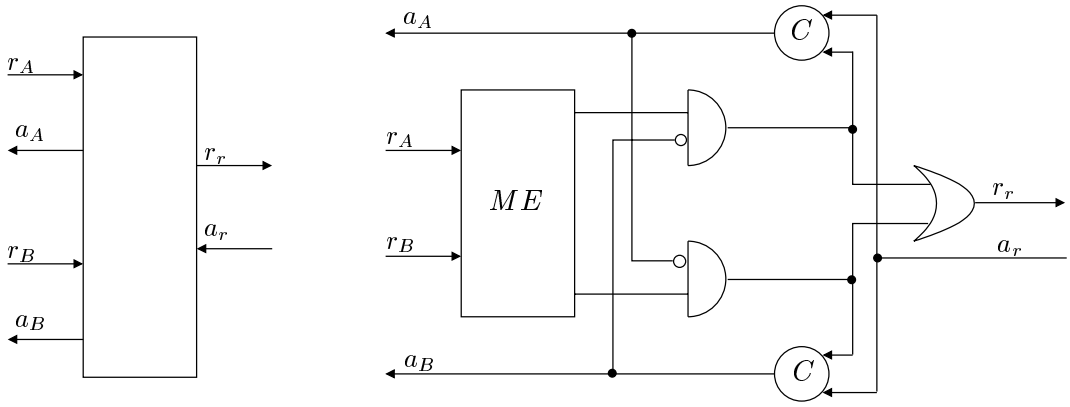
(a) (b)

Figure 1.22: The mutual exclusion element: (a) block diagram; (b) CMOS implementation.

it evolves to a stable one. Also, it was proved that oscillatory and meta-stable anomalies are inevitable in binary logic implementations of arbiters. “Thus, the only possibility of designing correct arbiters remains in the development of a hybrid, rather than purely digital, unbounded implementation based on automatic locking of the anomalous behavior by means of analog circuits” [47].

The mutual exclusion (ME, mutex) element [72] is a two-input, two-output device which presents such behavior. An ME element is basically a latch with a meta-stability detector built-in. Its block diagram representation is given in figure 1.22.a, while in part b of the same figure is shown a possible CMOS implementation. The pairs of transmission gates after the latch form the meta-stability detector.

When both inputs are low, transistors T_1 and T_2 are off, transistors T_3 and T_4 are on, and thus, the outputs are both low. If one of the inputs, say r_A, raises, signal Q_A falls, turning on transistor T_2 and turning off T_4; then, output g_A goes to high. If now input r_B raises, Q_B remains high and no transition occurs at g_B. The interesting case, however, occurs when both inputs change from 0 to 1 almost simultaneously. The latch eventually enters a meta-stable state and the meta-stability detector enters into action. Signals Q_A and Q_B will differ by less than the threshold voltage and then transistors T_1 and T_2 will be off. Outputs g_A and g_B will remain low. When the latch part of the mutex has resolved the meta-stability, it will stabilize into one of its stable states. At this point, one of the outputs of the mutex goes high, while the other remains low.



(a) (b)

Figure 1.23: A 4-phase protocol, 2-input arbiter: (a) block diagram; (b) implementation using an ME element and 2 Muller C-elements.

With the ME element different types of arbiters with a correct behavior can be built. For example the arbiter in figure 1.23 [31] is a two-input, 4-phase protocol arbiter. It can be used to control the access of two concurrent processes to a common resource, connected to the r (r_r, a_r) interface. It can also be used as a building block to construct arbiters of higher order.

1.8 Overview

The main objectives of this work are focused on two issues. From one side we want to deal with specifications containing speed-independent conflict situations. These situations make the speed-independent implementation as a pure digital circuit impossible. The idea behind the work is to isolate such conflict situations and resolve them using special arbitration components. These components have a partial analog implementation, which captures the conflict, while delivering pure digital signals at their outputs. For instance, the mutual exclusion element, mentioned in the previous section and represented in figure 1.22, can fairly resolve the simultaneous occurrence of positive transitions at its inputs, while its outputs have pure digital values at all time.

From the other side we want to benefit from the power of existing tools for the synthesis of speed-independent asynchronous circuits. SIS [73] and *petrify* [18] are tools which support the synthesis

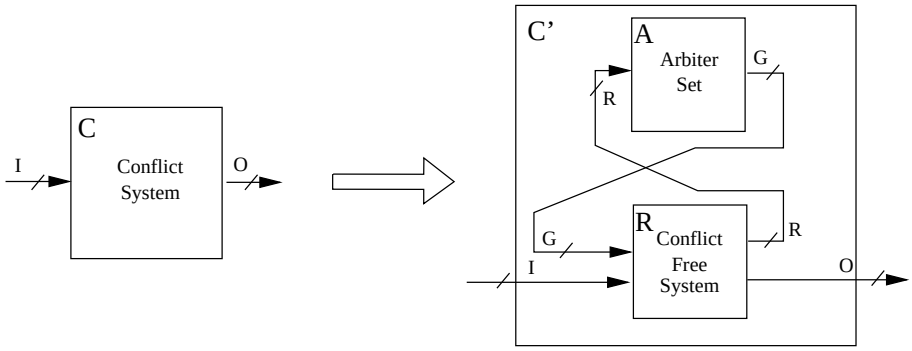


Figure 1.24: Overview of transformation process. The composite behavior of sub-systems $\mathcal{A}$ and $\mathcal{R}$ must be equivalent to the original behavior $\mathcal{C}$.

of speed-independent asynchronous circuits. Thus, our purpose is to develop a method which takes a specification with conflicts as input data, transforms this specification in order to resolve the conflicts with the special arbitration components and delivers a transformed specification suitable to be synthesized with the existing tools. The transformed specification generates the signals which "attack" the inputs of the arbiters and receives as inputs their outputs. An overview of this transformation process is depicted in figure 1.24. If $\mathcal{C}$ is a system specification with conflicts, it must be transformed into two sub-systems, $\mathcal{A}$ and $\mathcal{R}$, $\mathcal{A}$ being a set of arbiters and $\mathcal{R}$ being a conflict-free specification, suitable to feed the existing synthesis tools.

The original specification can be given either as a signal transition graph or a state graph description. Thus, the conflict analysis and managing can (must) be done in the two domains. An attempt was done to manage conflicts directly at STG level [68]. However some difficulties found at this level, for instance, problems of conflict localization due to dummy events, made us deal with conflicts at state level first. The fact that we have planned to use a tool which synthesizes through the intermediate state graph model, also reduces the penalty of working at the state level.

Two types of conflicts are identified: non-persistences and non-commutativities. The former are characterized by the disabling of some events because of the occurrence of some other events. In the latter, the order of occurrence of simultaneously enabled events make the system evolve in different directions. Conflicts of both types are analyzed, but non-persistences are deeply studied. Conflicts of this type are associated to exclusion relations among regions, regions being sets of states entered by

the occurrence of some events and exited by the occurrence of others. The simplest form of exclusion relation corresponds to the mutual exclusion between two regions, meaning that the system can only be inside one of them at a time. When the system reaches a state belonging to the input borders of two mutual exclusive regions, it has to choose which one to enter. Under certain circumstances this can cause anomalous behavior and thus a conflict exists. Many conflicts cannot be associated with the simplest mutual exclusion relation between two regions. But, they can be associated with a more general exclusion relation, that we call an exclusion relation of k out of n, meaning that the system cannot be inside more than k among n different regions.

A methodology is proposed to deal with specifications containing conflicts of the non-persistence type. First of all, a conflict point must be identified. A conflict point corresponds to a set of regions in an exclusion relation of k out of n. Second, an arbitration device is chosen to control the conflict point. The mutual exclusion (mutex) element presented in the previous section is a suitable device to control access to two regions under mutual exclusion. For the more general exclusion relation of k out of n, we proposed a new device, called genex $n \times k$. It can be built up using mutex elements. Third, the specification is transformed in order to transfer the conflict point to the arbiter. This three steps are repeated while there are conflicts remaining.

A set of tools was developed that aids in performing the task of implementing the transformation process. Instead of developing a unique integrated tool, we have decided to build different tools, each implementing a specific task. In this way we can access and analyze intermediate results and test different alternatives. The data for the examples presented along the thesis was obtained using these tools.

1.9 Structure of the Thesis

Apart from this chapter the thesis is organized in 5 chapters. In chapter 2 the main formalisms used to represent and manipulate circuit descriptions are given. First, transition systems and state graphs are presented. They determine the specification of a system (circuit) as a set of states and a set of transitions, which make the system evolve from a state to another. Then, Petri nets and signal transition graphs are presented. Now a system (circuit) is represented by the causality, concurrency and choice relations between its events. Regions, a topic central to the thesis that makes the connection between

state and event models, concludes the chapter.

The transformation process takes place at state level, where transition systems and state graphs are the used formalisms. Chapter 3 is dedicated to the introduction of a set of transformations and operations realized on transition systems and state graphs. Projection is presented as an operation which hides one or more events when applied to a transition system and hides one or more signals when applied to state graphs. The product of transition systems and/or state graphs, the main operation of the transformation process is presented next. The chapter concludes with two important transformations for the synthesis flow: insertion of new events on transition systems and insertion of new signals on state graphs.

The study of conflicts is done in chapter 4. Both types, non-persistences and non-commutativities, are covered. The methodology to deal with specifications containing conflicts of the former type is also studied in this chapter. The notion of exclusion relation of k out of n plays a central role in this methodology.

Chapter 5 is dedicated to implementation aspects necessary to build a synthesis procedure. Only synthesis of non-persistence specifications is covered. A mathematical formalism to describe the transformation process is developed. The set of tools developed during the work supporting the thesis is also presented in chapter 5.

Finally, chapter 6 concludes the thesis, summarizing the major results achieved and pointing out directions for future research.

Chapter 2

Graph Models

Graph models are a simple mathematical formalism that nicely describe the behavior of systems. A system that evolves over time is often described by a finite state automaton, formed of states and transitions between these states. This finite state automaton can be represented by a directed graph, or more generally a directed multi-graph, where vertices represent states and arcs represent transitions between states. Transitions are often bound to the occurrence of some event or action. This corresponds to labeling arcs of the graph with the events or actions.

State graphs are the graph model normally used to represent speed-independent asynchronous circuits at state level. It can be seen as a specialization of the more general model called finite transition systems. In this chapter we introduce both finite transition systems and state graphs.

Systems can also be described by partially ordered sequences of events. Considering the case of asynchronous circuits this corresponds to the information captured by a timing diagram. Petri nets appear as the natural event-based graph model to capture such behavior. Petri nets are represented by directed graph with two types of nodes: transitions and places. Transitions are bound to events of the system. Places hold the pre- and post-conditions for the occurrence of the events. The asynchronous community, specially those involved with speed-independent design, use a slightly modified version of a Petri net called signal transition graph (STG).

There is a connection between an event-level description of a system and a behavior-equivalent state-level description, in the sense that it is possible to transform one into the other and vice-versa. The

theory of regions, presented at the end of this chapter, appears as a discipline that aids in making that connection.

2.1 Transition Systems

Systems are often represented by a set of states, the states of the system, and a set of transitions which make the system "move" from one state to another. The *transition system* is a formalism which allows such a representation of a system.

Transition systems have been defined in slightly different ways by different authors. See for instance [1, 51]. Our definition mostly follows [1].

Definition 2.1 (Transition System)
A transition system is a 4-tuple $\langle S, T, \alpha, \beta \rangle$ *where*

- S *is the set of states;*
- T *is the set of transitions;*
- $\alpha, \beta : T \rightarrow S$ *are functions which map each transition to its source and destination states, respectively.*

Sets S and T can be finite or infinite. In sequel we will only consider finite transition systems. Figure 2.1 shows two (finite) transition systems, each with 3 states and 5 transitions.

Consider the function $\langle \alpha, \beta \rangle : T \mapsto S \times S$ which maps every transition to the ordered pair of states they connect. This function is injective if no pair of different transitions have the same source and the same destination states. This is, for instance, the case of transition system S_A in figure 2.1. But, is not the case for system S_B, since $\langle \alpha, \beta \rangle(t_3) = \langle \alpha, \beta \rangle(t_4)$. Note that if $\langle \alpha, \beta \rangle$ is injective, the graph underlying the transition system is a *directed graph* (*digraph*); otherwise it is called a directed *multi-graph* [56].

If function $\langle \alpha, \beta \rangle$ is injective, the transition system is completely defined by the tuple $\langle S, \Theta \rangle$, where $\Theta = \langle \alpha, \beta \rangle(T) \subseteq S \times S$ is called the *transition relation*. For instance, transition system S_A of

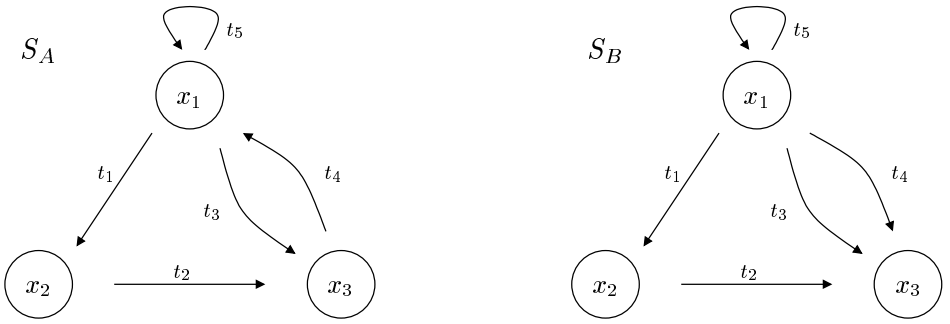


Figure 2.1: Two simple transition systems. They are almost equal except for transitions t_4 which have different directions.

figure 2.1 is completely defined by the sets

$$S = \{x_1, x_2, x_3\}$$
$$\Theta = \{\langle x_1, x_2\rangle, \langle x_2, x_3\rangle, \langle x_3, x_1\rangle, \langle x_1, x_1\rangle, \langle x_1, x_3\rangle\} \ .$$

2.1.1 Labeled Transition Systems

If transition systems are used to represent the behavior of systems, transitions are usually interpreted as the occurrence of actions or events. The term labeled transition system is used to refer to such a transition system.

Definition 2.2 (Labeled Transition System)
A labeled transition system *is the tuple* $G = \langle S, T, \alpha, \beta, E, \lambda\rangle$*, where*

- $\langle S, T, \alpha, \beta\rangle$ *is a transition system;*
- E *is the set of actions or events (usually called the alphabet);*
- $\lambda : T \to E$ *is a function which maps every transition* $t \in T$ *to an action or event on* E*.*

Similarly to what was done before, one can define function $\langle \alpha, \lambda, \beta\rangle : T \to S \times E \times S$ which maps every transition $t \in T$ to a 3-tuple composed of the source state, the event and the destination state of t. One can assume that $\langle \alpha, \lambda, \beta\rangle$ is injective. Indeed, it makes no sense to assume that two different transitions, triggered by the same event, both make the transition system move from the same

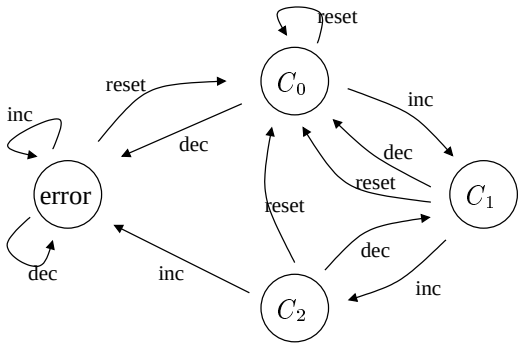


Figure 2.2: A labeled transition system describing the behavior of an up-down counter ranging from 0 to 2, with overflow and underflow detection.

source state to the same destination state. So a labeled transition system can be defined by the tuple $\langle S, E, \Theta \rangle$, where $\Theta = \langle \alpha, \lambda, \beta \rangle(T)$. This leads to a second definition of labeled transition system.

Definition 2.3 (Labeled Transition System)

A labeled transition system is a tuple $G = \langle S, E, \Theta \rangle$, where

- *S is a set of states;*
- *E is a set of events;*
- *$\Theta \subseteq S \times E \times S$ is the transition relation.*

An element $\langle s, e, s' \rangle \in \Theta$ is often also denoted by $s \mapsto e \to s'$ or $s \xrightarrow{e} s'$.

Figure 2.2 shows a labeled transition system, based on an example from [1]. It represents an up-down counter ranging from 0 to 2. It enters an error state if decremented at 0 or incremented at 2. Its definition sets are:

$$
\begin{aligned}
S &= \{C_0, C_1, C_2, \text{error}\} \\
E &= \{\text{inc}, \text{dec}, \text{reset}\} \\
\Theta &= \{\langle C_0, \text{inc}, C_1 \rangle, \langle C_0, \text{dec}, \text{error} \rangle, \langle C_0, \text{reset}, c_0 \rangle, \\
&\quad \langle C_1, \text{inc}, C_2 \rangle, \langle C_1, \text{dec}, C_0 \rangle, \langle C_1, \text{reset}, C_0 \rangle, \\
&\quad \langle C_2, \text{inc}, \text{error} \rangle, \langle C_2, \text{dec}, C_1 \rangle, \langle C_2, \text{reset}, C_0 \rangle, \\
&\quad \langle \text{error}, \text{inc}, \text{error} \rangle, \langle \text{error}, \text{dec}, \text{error} \rangle, \langle \text{error}, \text{reset}, C_0 \rangle\}
\end{aligned}
$$

Often, given a transition $t = \langle s, e, s' \rangle$, it is necessary to isolate either the source state, or the destination state, or the event. We will respectively use functions α, β, and λ for that purpose.

Note that with definition 2.3, a transition system can be *nondeterministic*. In a given state, the same event can label two different transitions leading to different states, that is, it is possible to have two transitions t_1 and t_2 such that $\alpha(t_1) = \alpha(t_2)$, $\lambda(t_1) = \lambda(t_2)$, but $\beta(t_1) \neq \beta(t_2)$.

Often systems have a fixed initial state.

Definition 2.4 (initialized labeled transition system)
A labeled transition system where a state is defined as being the initial state is called an initialized labeled transition system. *It is defined by the tuple* $\langle S, E, \Theta, s_0 \rangle$, *where* $s_0 \in S$ *is the initial state.*

2.1.2 Sequences

The following definitions are inherited from graph theory. An alternating sequence

$$\omega = \langle x_0, t_1, x_1, t_2, \cdots, t_n, x_n \rangle$$

of states x_i and transitions t_i such that $t_i = \langle x_{i-1}, e_i, x_i \rangle$, for $i = 1, 2, \cdots, n$, is called a *walk*. The states x_0 and x_n are called the *outer states* of the walk, x_0 being the *initial* state and x_n the *final* one; the other states are called *inner* or *intermediate*. Associated to each state x we also define the *empty walk* which starts and ends in x and contains no transition. In a transition system, there are as many empty walks as states. If in a walk all the transitions are different it is called a *chain*. If, additionally, all the states are different, with the possible exception of the outer ones, it is called a *path*. If states x_0 and x_n are the same, the walk, chain, or path is called *cyclic*; otherwise it is called *noncyclic*. A cyclic path is also called a *contour*.

In transition system of figure 2.2 the following sequences are walks:

$$\omega_1 = \langle C_1, \langle C_1, \text{dec}, C_0 \rangle, C_0, \langle C_0, \text{inc}, C_1 \rangle, C_1, \langle C_1, \text{reset}, C_0 \rangle, C_0 \rangle$$
$$\omega_2 = \langle C_0, \langle C_0, \text{inc}, C_1 \rangle, C_1, \langle C_1, \text{inc}, C_2 \rangle, C_2 \rangle$$
$$\omega_3 = \langle C_0, \langle C_0, \text{inc}, C_1 \rangle, C_1, \langle C_1, \text{inc}, C_2 \rangle, C_2, \langle C_2, \text{reset}, C_0 \rangle, C_0 \rangle$$

Moreover, all of them are also chains, but only ω_2 and ω_3 are paths. Finally, path ω_3 is cyclic, so it is a contour.

A transition is a triplet containing its source and destination states. So, a walk can be represented just by the sequence of transitions

$$\omega = \langle t_1, t_2, \cdots, t_n \rangle$$

However, this notation makes it necessary to define a way to represent the empty walk, since there is an empty walk for each state of the transition system: ε_x is used to denote the empty walk associated to state x.

The length of a sequence is its number of transitions. If it is infinite the walk is called *infinite*; otherwise it is called *finite*. Let Θ be the transition relation of a transition system. The sets of all finite and infinite walks definable over Θ are denoted respectively by Θ^+ and Θ^ω. The set of all finite walks, including all the empty walks, is denoted by Θ^* and represents the *transitive closure* of the transition relation Θ.

Let $\omega = \langle t_1, t_2, \cdots, t_n \rangle$ be a finite walk. The initial state of a walk coincides with the source state of the first transition of that walk. Similarly, the final state of a walk coincides with the destination state of the last transition of the walk. Thus, functions α and β, used to return respectively the source and destination states of a transition, can be extended to walks in the following way:

$$\begin{aligned} \alpha(\omega) &= \alpha(t_1) \\ \beta(\omega) &= \beta(t_n) \end{aligned}$$

Given a transition system, consider that when one goes from one state to another each transition can be crossed either in the forward or in the backward direction. A sequence defined in this way is called a semi-walk. Formally, an alternating sequence

$$\omega = \langle x_0, t_1, x_1, t_2, \cdots, t_n, x_n \rangle$$

of states and transitions, such that transitions t_i are either of the form $t_i = \langle x_{i-1}, e_i, x_i \rangle$ or $t_i = \langle x_i, e_i, x_{i-1} \rangle$, for $i = 1, 2, \cdots, n$, is called a *semi-walk*. For semi-walks we can not simplify notation omitting states. *Semi-chain*, *semi-path*, and *semi-contour* are defined in a similar way.

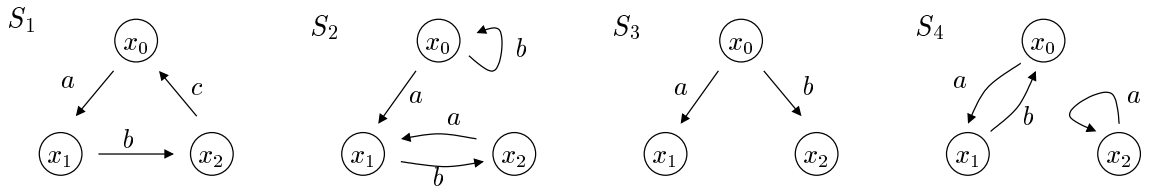


Figure 2.3: Different type of transition systems from a reachability point of view: (a) strongly connected; (b) unilaterally connected; (c) weakly connected; (d) not connected.

2.1.3 Reachability

A state x_v is *reachable* from a state x_u if there is a walk ω connecting x_u to x_v, that is $\alpha(\omega) = x_u$ and $\beta(\omega) = x_v$. If, in addition, x_u is also reachable from x_v, x_u and x_v are said to be *mutually reachable*. Since the empty walk is defined, every state is reachable from itself. Moreover, $\alpha(\varepsilon_x) = \beta(\varepsilon_x) = x$. Θ^*, which, as already mentioned, represents the *transitive closure* of the transition relation Θ, represents also the *reachability relation* between states.

A transition system is called *strongly connected* if any two different states are mutually reachable. It is called *unilaterally connected* if for any two states x_u and x_v, x_u is reachable from x_v or x_v is reachable from x_u. Finally it is *weakly connected* if any two vertices are connected by a semi-walk.

Figure 2.3 shows different types of transition systems from a reachability point of view. System S_1 is strongly connected. System S_2 is unilaterally connected; state x_0 is not reachable from neither x_1 nor x_2. System S_3 is weakly connected; neither x_1 is reachable from x_2, nor x_2 is reachable from x_1. However, there is a semi-walk connecting them. Finally system S_4 is not connected; state x_2 is completely disconnected from states x_0 and x_1.

2.1.4 Transition Sub-system

Often instead of dealing with the whole transition system we need to refer to only part of it. A part here refers to a subset of its states and a subset of its transitions. The following definitions are also inherited from digraph theory. Let $G = \langle S, E, \Theta, s_0 \rangle$ be an initialized labeled transition system. A *transition sub-system* of G is another transition system $G' = \langle S', E', \Theta', s_0 \rangle$ where $S' \subseteq S$, $E' \subseteq E$, and $\Theta' \subseteq \Theta$. Note that being a transition system it also means $\Theta' \subseteq S' \times E' \times S'$. Let

$X \subseteq S$. The transition sub-system of G *induced* by X, denoted by $G[X]$, is a transition sub-system $G' = \langle S', E', \Theta', s_0 \rangle$ such that $S' = X$ and $\langle s, e, s' \rangle \in \Theta' \iff (s, s' \in S' \wedge \langle s, e, s' \rangle \in \Theta)$.

A *strong component* of a transition system is a maximal under inclusion strong transition sub-system. Similarly, A *weak component*, or simply *connected component*, of a transition system is a maximal under inclusion weak transition sub-system.

2.1.5 Traces

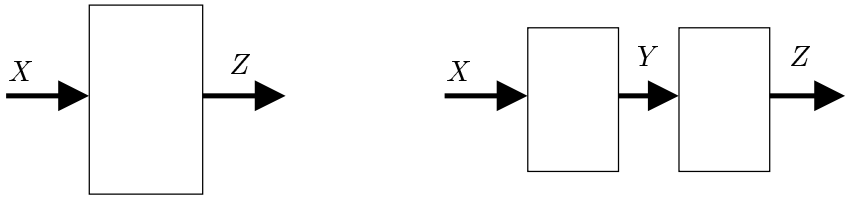
Let A be a set which we call the *alphabet*. Consequently its elements are called *letters* ([11]) and any sequence of letters from A is called a *word*. Also the word of length 0, the *empty word*, is usually considered and denoted by ε. The set of all unbounded above finite length words over alphabet A, including the empty one, is denoted by A^*. Any subset of A^* is called a *language* over alphabet A.

Let now $G = \langle S, E, \Theta, s_0 \rangle$ be a transition system and let $\omega = \langle t_1, t_2, \cdots, t_n \rangle$, for $t_i \in \Theta$, be a finite walk, that is, $\omega \in \Theta^*$. The sequence $\tau = \langle \lambda(t_1), \lambda(t_2), \cdots, \lambda(t_n) \rangle$ represents a sequence of events (letters) from alphabet E, and is usually called a *trace*. Thus, a trace is a word over alphabet E, feasible in system G. The set of all different traces defined by Θ^* is the language accepted by transition system G, and is denoted by $\mathcal{L}(G)$.

2.2 State Graphs

Let an asynchronous circuit be represented by a black box with a set of input and a set of output signals (see figure 2.4.a). Its behavior can be described by a finite state machine, which changes state whenever a transition occurs at one of its input or output signals. Therefore it can be represented by a labeled transition system, where each transition is bounded to the rising or the falling transition of one of its signals. The label representing a signal transition is composed of a signal name and a sign, indicating the direction of the transition. The rising and the falling transitions are represented respectively by the plus ($+$) and minus ($-$) signs. Also the asterisk ($*$) sign is used to refer to any of the transitions.

In addition to the input and output signals, internal signals can also be used in the specification of a circuit behavior. The existence of the internal signals can come from an initial partition of the



(a) (b)

Figure 2.4: Black boxes representing asynchronous circuits with a set X of input and a set Z of output signals.

global behavior we want to describe. For instance, we can eventually decompose our black box from figure 2.4.a in the two communicating black boxes of figure 2.4.b. The set of signals Y is internal to the global system. Internal signals can also appear due to transformations of some transition system.

We will use the term *state graph* to designate a transition system with the transition interpretation given above. We are aware it is abusive, since the common definition of state graph includes a state interpretation [17]. However we believe no confusion arises. The common state interpretation can be derived from the transition interpretation, and so we will use the same term for both definitions. This bad use has also been used somewhere else. For instance, the synthesis package *petrify*[18] use a description format called *state graph* where the state interpretation is not explicitly given. It is generated by a tool during the synthesis procedure.

Definition 2.5 (state graph)

An initialized labeled transition system $\langle S, E, \Theta, s_{in} \rangle$ *is called a* state graph *if*

- $E = V \times \{+, -\}$*; and*
- $V = V_O \cup V_I$ *is a set of signals, with* V_I *the set of input signals,* V_O *the set of output and internal signals, and* $V_I \cap V_O = \emptyset$*.*

2.2.1 Switch Count Correctness

If a state graph description does represent a circuit behavior, a constraint is imposed to the order of events of the same signal. No signal can switch twice in a given direction (either rising or falling)

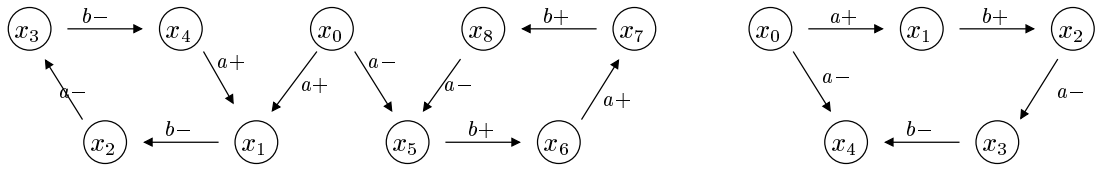


Figure 2.5: Two switch-over correct transition systems that can not represent a circuit description.

without switching in the opposite direction in between. Thus, for every trace (sequence of events), transitions of the same signal must alternate in their signs. This property is called *switch-over correctness* [47], and is a necessary condition for the realizability of a description.

Definition 2.6 (switch-over correctness)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ *be a state graph and let* $\tau = \langle e_1, e_2, \cdots, e_n \rangle$ *be a trace in* G*. Trace* τ *is* switch-over correct *if*

$$\forall_{e_i, e_j \in E}, \quad [(i < j) \wedge (e_i = e_j)] \Rightarrow [\exists_k : (i < k < j) \wedge (e_k = \neg e_i)]$$

where the operator $\neg$ *is used to represent a signal transition with the same name and the opposite sign than the one it is applied to. (For instance,* $\neg v+ = v-$*.)*

A state graph is switch-over correct *if all of its traces are switch-over correct.*

However it is not sufficient. Consider for instance the transition systems (state graphs) in figure 2.5. For every feasible trace, transitions of the same signal alternate in their sign, that is, they are switch-over correct. However none of these transition systems can represent a circuit behavior. In both cases, if transition $a+$ is enabled in state x_0, then signal a must be at 0 in that state. But transition $a-$ is also enabled in the same state, and so signal a must be at 1, which leads to a contradiction.

The state graph on the left side is weakly connected, while the one on the right side is unilaterally connected. If a state graph is strongly connected, switch-over correctness does guarantee the absence of this kind of contradiction. This is stated by the following lemma.

Lemma 2.7

Let $G = \langle S, E, \Theta, s_{in} \rangle$ *be a state graph. If* G *is strongly connected and switch-over correct, then both the rising and falling transitions of any signal cannot be enabled at the same state.*

Proof

Let a be a signal, s a state and assume $\langle s, a+, s_+\rangle, \langle s, a-, s_-\rangle \in \Theta$, that is $a+$ and $a-$ are enabled in s. Assuming G is strongly connected, then there is a cycle ω beginning on s, passing on s_+ and ending on s. Also assuming G is switch-over correct, then the signal transition $a+$ occurs in ω as many times as $a-$. Also, the occurrences of both events are interleaved. Thus, moving backward in ω, starting on s, we will find a $a-$ before finding a $a+$. But that signal transition is followed by transition $\langle s, a-, s_-\rangle$ without an $a+$ in between, contradicting the switch-over correctness assumption.

At this point we are facing a dilemma: either we limit acceptable specifications to strongly connected state graphs, or we define a property more general than switch-over correctness. We will follow the latter approach. Consistency, a property defined on states and presented below, can be used for that purpose. However, we are introducing a new property, that similarly to switch-over correctness is based on events. But it is defined on semi-walks instead of walks.

Definition 2.8 (switch count)

Let $G = \langle S, E, \Theta, s_0\rangle$ be a state graph, with $E = V \times \{+, -\}$; let $\omega = \langle x_0, t_1, x_1, \cdots, x_{n-1}, t_n, x_n\rangle$ be a semi-walk in G, that is, each t_i has the form of either $\langle x_{i-1}, e_i, x_i\rangle$ or $\langle x_i, e_i, x_{i-1}\rangle$; finally, let $v \in V$. The switch count *of v over ω, denoted* **cnt**(v, ω)*, is defined as the result of the following procedure:*

let $cnt = 0$;
for $i = 1$ **to** n **do**
  if $t_i = \langle x_{i-1}, e_i, x_i\rangle$ **and** $e_i = v+$ **then** $cnt = cnt + 1$;
  if $t_i = \langle x_{i-1}, e_i, x_i\rangle$ **and** $e_i = v-$ **then** $cnt = cnt - 1$;
  if $t_i = \langle x_i, e_i, x_{i-1}\rangle$ **and** $e_i = v+$ **then** $cnt = cnt - 1$;
  if $t_i = \langle x_i, e_i, x_{i-1}\rangle$ **and** $e_i = v-$ **then** $cnt = cnt + 1$;
return *cnt;*

Moving a transition associated with signal v forwards, the count is incremented for the rising transition and decremented for the falling transition. Moving a transition backwards, the incrementing/decrementing is done in the opposite direction.

Definition 2.9 (switch count correctness)

1. *A signal v is* switch count correct *on state graph G if for every semi-walk ω defined on G,* $\mathbf{cnt}(v, \omega) \in \{-1, 0, 1\}$*;*

2. *state graph G is* switch count correct *if all its signals are switch count correct.*

State graphs of figure 2.5 are not switch count correct. For the one on the left side, semi-walk $\langle x_1, \langle x_0, a+, x_1\rangle, x_0, \langle x_0, a-, x_5\rangle, x_5\rangle$ has a switch count of -2. For the one on the right side, semi-walk $\langle x_1, \langle x_0, a+, x_1\rangle, x_0, \langle x_0, a-, x_4\rangle, x_4\rangle$ also has a switch count of -2.

Let ω_1 and ω_2 be two semi-walks of a switch-count correct state graph, such that the ending state of ω_1 is the beginning state of ω_2. Their composition, denoted by $\omega_1\omega_2$, is the semi-walk obtained following ω_1 by ω_2. The following lemma is derived straightforward from definition 2.8, and so is given without a proof.

Lemma 2.10

$$\mathbf{cnt}(v, \omega_1\omega_2) = \mathbf{cnt}(v, \omega_1) + \mathbf{cnt}(v, \omega_2)$$

Let ω^{-1} be the reverse of ω, that is, if $\omega = \langle x_0, t_1, x_1, \cdots, x_{n-1}, t_n, x_n\rangle$, then $\omega^{-1} = \langle x_n, t_n, x_{n-1}, \cdots, x_1, t_1, x_0\rangle$. The following lemma is also derived straightforward from definition 2.8, and so is also given without a proof.

Lemma 2.11

$$\mathbf{cnt}(v, \omega) + \mathbf{cnt}(v, \omega^{-1}) = 0$$

Let ω_1 and ω_2 be two semi-walks of a switch-count correct state graph, with the same beginning state and the same ending state.

Lemma 2.12

$$\mathbf{cnt}(v, \omega_1) = \mathbf{cnt}(v, \omega_2)$$

Proof

Assume $\mathbf{cnt}(v, \omega_1) \neq \mathbf{cnt}(v, \omega_2)$*. Since, by lemma 2.11,* $\mathbf{cnt}(v, \omega_2) = -\mathbf{cnt}(v, \omega_2^{-1})$*, we have* $\mathbf{cnt}(v, \omega_1) + \mathbf{cnt}(\omega_2^{-1}) \neq 0$*. The ending state of ω_1 coincides with the beginning state of ω_2^{-1}. Thus, $\omega_1\omega_2^{-1}$ is a semi-walk and has a switch count different from 0. Also, the beginning state of ω_1 coincides with the ending state of ω_2^{-1}, and $\omega_1\omega_2^{-1}\omega_1\omega_2^{-1}$ is also a semi-walk. Its switch count is necessarily lower than -1 or greater than 1, contradicting the switch-count correct assumption.*

2.2.2 State Graph

The purpose of circuit specification is to make circuit synthesis. In that sense, states must be encoded in a binary vector, where each bit represents the value 0 or 1 of a signal. Eventually, the signals used for specification are enough to differently encode all states. Sometimes, however, it is necessary to add some extra internal signals.

The state graph description given by definition 2.5 is normally associated with a state labeling function, which assigns a binary vector of length n to each of its states. There is a bit in this vector for each signal of the state graph. The usual state graph definition includes this state labeling function.

Definition 2.13 (state graph)
A state graph *is the tuple* $\langle S, E, \Theta, \lambda_S, s_{in} \rangle$*, where*

- $\langle S, E, \Theta, s_{in} \rangle$ *is an initialized labeled transition system;*
- E*, the set of events, is formed from the product* $V \times \{+, -\}$*, where* V *is the set of signals;*
- $\lambda_S : S \times V \mapsto \{0, 1\}$ *is a state labeling function, which assigns a binary value for each* $s \in S$ *and for each* $v \in V$*.*

Often all the signal bits associated with the same state are grouped together in a binary vector. That is, the state labeling function takes the form $\lambda_S : S \mapsto \{0, 1\}^n$, which involves an implicitly ordering of the signal bits. In the typical graphical representation of a state graph, states appear decorated with this binary vector. Moreover, bits of excited signals appear further decorated with a prime[1]. See as an example the state graph depicted in figure 1.20.

2.2.3 Consistency

If a state graph, as defined by definition 2.13, does represent a circuit behavior, there is a constraint in its state labeling function. For instance, if signal transition $v+$ is enabled in state x, the state bit associated with v must 0 at x and 1 at all states reached after the firing of $v+$. This property of state graphs is called *consistency*.

[1] An asterisk is also used for the same purpose.

Definition 2.14 (consistency)

A state graph $G = \langle S, E, \Theta, \lambda_S, s_{in} \rangle$, with $E = V \times \{+, -\}$, is called consistent *(has a* consistent state assignment*) if λ_S is such that for each transition $\langle s, e, s' \rangle \in \Theta$, and for each signal $v \in V$,*

- **if** $e = v+$, **then** $\lambda_S(s, v) = 0$ **and** $\lambda_S(s', v) = 1$
- **if** $e = v-$, **then** $\lambda_S(s, v) = 1$ **and** $\lambda_S(s', v) = 0$
- **otherwise** $\lambda_S(s, v) = \lambda_S(s', v)$

Existence of a consistent state assignment is closed related to switch count correctness as is established by the following theorem.

Theorem 2.15

The necessary and sufficient condition for a state graph to accept a consistent state assignment is to be switch count correct.

Proof

Necessity: Let $G = \langle S, E, \Theta, \lambda_S, s_{in} \rangle$, with $E = V \times \{+, -\}$, be a consistent state graph. Choose at random a semi-walk $\omega = \langle x_0, t_1, x_1, \cdots, x_{n-1}, t_n, x_n \rangle$ and a signal v from G, and determine the switch count of v in ω. Let $\langle x_{i-1}, t_i, x_i \rangle$ be an excerpt of ω, with one transition. Transition t_i can have the form $\langle x_{i-1}, e, x_i \rangle$ or $\langle x_i, e, x_{i-1} \rangle$.[2] If $\lambda_S(x_{i-1}, v) = \lambda_S(x_i, v)$, the label of t_i is not associated with signal v and thus no increment or decrement is done when moving from x_{i-1} to x_i. If $\lambda_S(x_{i-1}, v) = 0$ and $\lambda_S(x_i, v) = 1$, t_i has the form $\langle x_{i-1}, v+, x_i \rangle$ or $\langle x_i, v-, x_{i-1} \rangle$. In both cases going from x_{i-1} to x_i makes the switch count increments. Similarly if $\lambda_S(x_{i-1}, v) = 1$ and $\lambda_S(x_i, v) = 0$, the switch count decrements. No two increments (decrements) can exist without a decrement (increment) in between, and so **cnt**$(\omega) \in \{-1, 0, 1\}$.

Sufficiency: Assume G be switch-count correct. Let v be a signal and let $\langle s, v+, s' \rangle \in \Theta$. Let $\gamma(x, v)$ be the switch count of v over every semi-walk beginning on s and ending on x. Note that by lemma 2.12 all semi-walks with the same beginning and the same ending states have the same switch count. By definition 2.8, $\gamma(s, v) = 0$ and $\gamma(s', v) = 1$. Moreover, $\gamma(x, v) \in \{0, 1\}$. Indeed, by definition 2.9 and the assumption of switch-count correctness,

[2] Remember we talking about semi-walks.

$\gamma(x, v) \in \{-1, 0, 1\}$; but it cannot be -1, because in that case, by lemma 2.10, the semi-walk $\omega' = \langle s', \langle s, v+, s' \rangle, s, \cdots, x \rangle$ would have a switch count of -2.

For any transition $\langle x_1, v+, x_2 \rangle$ we have $\gamma(x_1, v) = 0$ and $\gamma(x_2, v) = 1$. Otherwise, if $\gamma(x_1, v)$ was equal to 1, $\gamma(x_2, v)$ would be equal 2. Similarly, for any transition $\langle x_1, v-, x_2 \rangle$ we have $\gamma(x_1, v) = 1$ and $\gamma(x_2, v) = 0$. Finally, for any transition $\langle x_1, e, x_2 \rangle$, with $e \notin \{v+, v-\}$, we have $\gamma(x_1, v) = \gamma(x_2, v)$.

Thus, $\lambda_S(x, v) = \gamma(x, v)$ is a state labeling function satisfying all conditions of definition 2.14.

Consistent state assignment is a necessary condition for deriving logic functions for signals encoding a state graph [16]. This encoding is required to be unambiguous, that is, a state code must uniquely represent a state of the specification. The first approach is to impose different states to have different binary vectors. In such a case the state graph is said to hold the *unique state code* (USC) property. The state graph of the D-latch depicted in figure 1.20 holds the unique state coding property.

But input signals are delivered by the environment. The circuit is responsible to generate the logic functions for output and internal signals. Therefore the unambiguous state assignment must concern only these signals. As we have already mentioned in the chapter 1, the next-state function for a signal v maps the state code of each state s into:

- 1 if signal v is stable at 1 or excited to go to 1 in s;
- 0 if signal v is stable at 0 or excited to go to 0 in s;

Additionally, all state codes that do not correspond to any reachable state from the initial state are mapped into a don't care. Thus, different states can have the same state code as long as they have the same set of excited output and internal signals. This property of state graphs is called *complete state coding* (CSC). It was shown [16] that a consistent state graph holding the complete state coding property has well-defined next-state functions for all output and internal signals. In figure 2.6 is depicted a state graph, borrowed from [17], with 10 states but only 8 different state codes. Let a and b be output signals and c and d input signals. States x_0 and x_2 have the same binary code 0110. Output signal b is excited in state x_0 but not in state x_2. Therefore there is a CSC violation and states x_0 and x_2 are said to be in CSC conflict. States x_5 and x_7 also have the same state code 1111, but they are not in conflict, since only input signals are excited in those states.

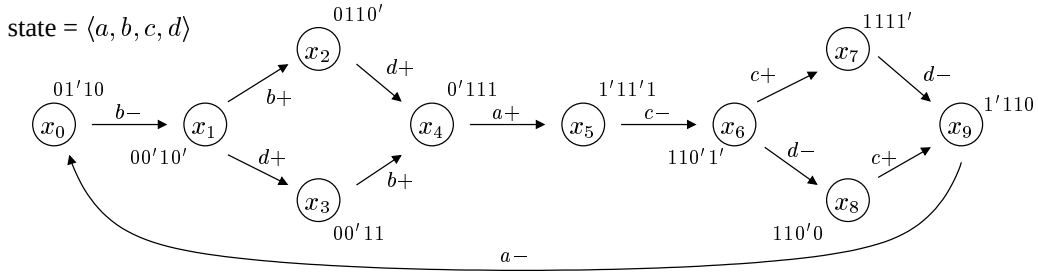


Figure 2.6: A state graph that does not hold the complete state coding property.

2.3 Petri Nets

A *Petri net* [61, 69] is a graphical and mathematical modeling formalism frequently used to describe the behavior of systems. The underlying structure of a Petri net is a directed, weighted, bipartite graph, with two kind of vertices, called *transitions* and *places*. Transitions and places are connected through arcs. Any arc can be either from a place to a transition or from a transition to a place. Arcs can never directly connect two transitions or two places. It is assumed there can exist only one arc connecting the same ordered pair of vertices. However, each arc is decorated with a label, called the weight, which is a positive integer. An arc with weight n is equivalent to the existence of n parallel arcs.

Definition 2.16 (Petri net structure)
A Petri net structure *(PNS) [61] a 4-tuple* $N = \langle P, T, F, W \rangle$*, where*

- P *is a set of* places*;*
- T *is a set of* transitions*;*
- $F \subseteq (P \times T) \cup (T \times P)$ *is the* flow relation*; and*
- $W : F \mapsto \{1, 2, 3, \ldots\}$ *is a weight function which assigns a positive integer to each arc* $f \in F$*.*

Often W is the constant 1 function and the Petri net structure, said to be *ordinary*, is simply defined by the 3-tuple $N = \langle P, T, F \rangle$.

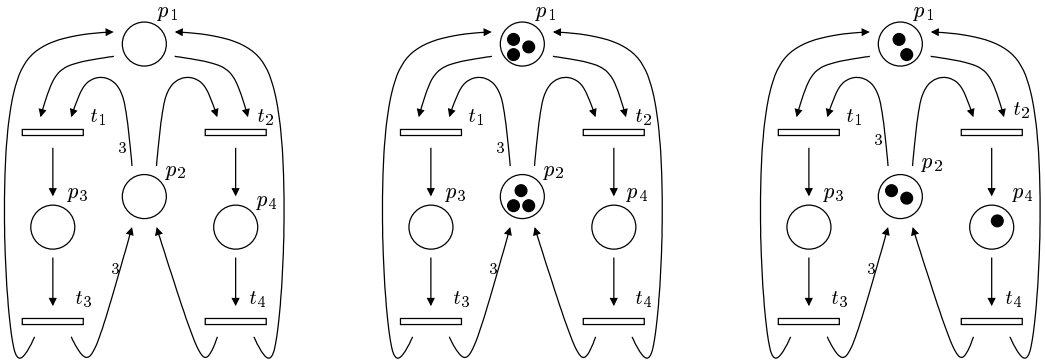


Figure 2.7: A Petri net structure and two Petri nets defined over it.

In the usual graphical representation places are represented by circles and transitions by bars or boxes. Weights are represented by numbers labeling the arcs. Arcs with weight 1 have usually the label omitted. Figure 2.7.a shows a Petri net structure with 4 places and 4 transitions.

A place $p \in P$ is a *predecessor* of a transition $t \in T$ if $\langle p, t \rangle \in F$. Likewise, a transition $t \in T$ is a *predecessor* of a place $t \in T$ if $\langle t, p \rangle \in F$. Successors can be defined in a similar way. The set of all predecessor and all successor places of a transition t are denoted by $\bullet t$ and $t\bullet$, respectively. Similarly, the set of all predecessor and all successor transitions of a place p are denoted by $\bullet p$ and $p\bullet$, respectively. In the Petri net structure of figure 2.7.a, transition t_1 has 2 predecessor places, p_1 and p_2, and 1 successor place, p_3. Thus, $\bullet t_1 = \{p_1, p_2\}$ and $t_1\bullet = \{p_3\}$.

Every place in a Petri net structure is *marked* with a non-negative integer. A *marking* is a function $M : P \mapsto \{0, 1, 2, \ldots\}$, which assigns a non-negative integer to each place $p \in P$. Graphically, the marking of each place is represented by a number or by black dots drawn inside the place. These black dots are usually referred to as *tokens*. A Petri net structure with an initial marking is called a *Petri net*.

Definition 2.17 (Petri net)

A Petri net *(PN) is a 2-tuple $N_M = \langle N, M_0 \rangle$, where N is a Petri net structure and M_0 is the initial marking.*

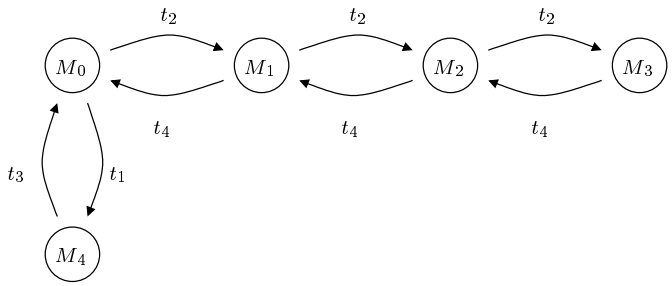


Figure 2.8: The reachability graph for the Petri net in figure 2.7.b.

Figure 2.7.b shows a Petri net obtained from the Petri net structure depicted in figure 2.7.a plus an initial marking which assigns 3 tokens to places p_1 and p_2 and 0 tokens to the other places.

The dynamics of a Petri net is defined as follows. A transition is *enabled* whenever all its predecessors places have a least as many tokens as the number labeling the connecting arcs. Any enabled transition can *fire*, and when it fires tokens are removed from every predecessor place and tokens are added to every successor place. The number of tokens removed/added depends on the arc labels. Referring to figure 2.7.b transition t_1 is enabled because both places p_1 and p_2 have at least one token each. Transition t_2 is also enabled, because place p_1 has at least one token and place p_2 has at least three tokens.

Whenever a transition fires the Petri net evolves from one marking to another. A marking M' is *reachable* from another marking M if there is a sequence of enabled transitions firing that produces M' starting from M. If, in the Petri net of figure 2.7.b t_2 fires, one token is removed from p_1, one token is removed from p_2 and one token is added to p_4. After the token flow, transition t_4 becomes enabled, transition t_1 becomes disabled and transition t_2 keeps enabled. This new scenario is represented by figure 2.7.c.

A marking of a Petri net represents a state of the system. *Playing the token game* we can construct a transition system where states correspond to markings and transitions correspond to transitions between markings. This transition system is called the *reachability (case) graph* of the Petri net. The reachability graph for the Petri net in figure 2.7.b is depicted in figure 2.8.

Petri nets are used to model the behavior of systems, which is done given interpretations to places and transitions. In a common interpretation, transitions are synchronized with events and places are used to represent pre-conditions and post-conditions for the occurrence of the events. For instance, when Petri nets are used to model the behavior of asynchronous digital circuits, transitions are synchronized with the falling or rising transitions of the signals of the circuit. Figure 1.21.a, in the Introduction, shows a labeled Petri net describing the behavior of a D-latch.

Definition 2.18 (labelled Petri net)
A labeled Petri net[3] *is a 4-tuple* $N_L = \langle N, M_0, E, \Lambda \rangle$*, where*

- $N = \langle P, T, F, W \rangle$ *is a Petri net structure;*
- M_0 *is the initial marking;*
- E *is an alphabet of labels (events); and*
- $\Lambda : T \longmapsto E$ *is the labeling function, which assigns a label (event) to each transition in the net.*

A labeled Petri net is called *single event* if Λ is bijective. For a single event Petri net, T can be replaced with E and Λ become the identity function. Thus, these labeled Petri nets can be represented by the tuple $\langle P, E, F, M_0 \rangle$.

A number of properties are important when we are using Petri nets for the modeling of asynchronous circuits:

Liveness. A marking M is *live* if, no matter what marking has been reached, any transition can be made enabled through some further firing sequence. A Petri net is live if its initial marking is live. The Petri net in figure 2.9.a is not live. If after a firing of transition t_1, transition t_3 fires a marking is reached with one token in place p_4 and zero tokens in all the other places. After this marking has been reached, it is impossible to evolve to a marking where transitions t_1 or t_2 are enabled. The Petri net in figure 2.9.b is live.

Boundedness and safeness. A Petri net is said to be k*-bounded* if the number of tokens that any place can hold after any firing sequence from the initial marking does not exceed a finite number k. A Petri net is said to be *safe* if it is 1-bounded. The Petri nets in figures 2.7.b, 2.9.a

[3]The terms *interpreted Petri net* or *synchronized Petri net* can be found in the literature with the same meaning.

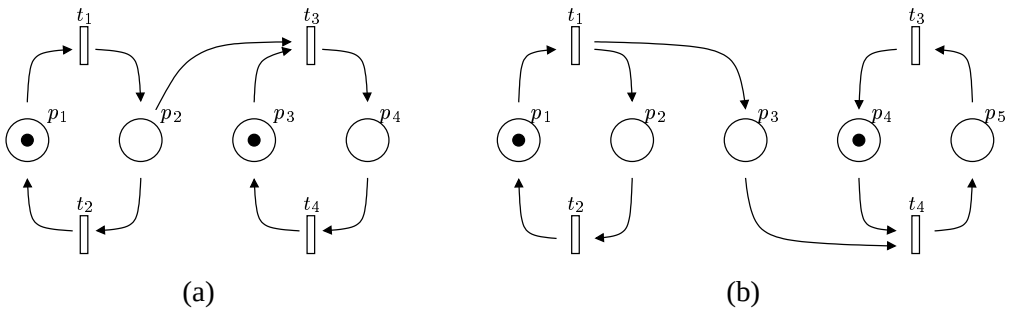


Figure 2.9: Illustrating liveness and boundedness in Petri nets: (a) bounded but not live; (b) live but not bounded. (Petri nets borrowed from [61].)

and 2.9.b are respectively 3-bounded, safe and unbounded. Boundedness is closely related to the existence of a finite implementation. If a Petri net is bounded, its reachability graph has a finite number of markings (states). On the contrary, an unbounded Petri net has an infinite reachability graph.

Persistence. A Petri net is *persistent* with respect to a transition t if for all markings M reachable from the initial marking M_0, such that t is enabled in M along with some other transition t', t remains enabled after the firing of t'. If the firing of t' brings the net to a marking M' where t is not enabled, t is said to be *disabled* by t', and the net is said to be *non-persistent* with respect to t. A Petri net is persistent if it is persistent with respect to all its transitions. The Petri net in figure 2.9.a is not persistent with respect to transitions t_2 and t_3. Consider the net evolves to the marking with one token in places p_2 and p_3 and zero tokens in places p_1 and p_4. Both transitions t_2 and t_3 are enabled. But the firing of one them will disable the other. The net in figure 2.9.b is persistent with respect to all transitions.

For labeled Petri nets we are often more interested in persistence of the events labeling the transitions than in persistence of the transitions themselves. We call this property *event persistence* or simply *persistence* if no confusion arises. For single event Petri net the two types of persistence are equivalent.

2.4 Signal Transition Graphs

As already mentioned in chapter 1, *signal transition graph* (STG) [16, 70] is the widest-used event-based model for the specification of the behavior of speed independent asynchronous circuits and of the environment it is immersed. An STG is a labeled Petri net where transitions are interpreted as events and places as pre- and post-conditions for the occurrence of those events. The classes of acceptable Petri nets mostly depend on the type of manipulation one wants to do. One restriction is, however, always imposed: the Petri net have to be bounded, in order to have a finite number of states, and so to be implementable. Another restriction is also usually considered: all arcs of the Petri net have unitary weight. Also the types of acceptable events have evolved since the early work on logic synthesis from STG. Nowadays the events considered are those related with the switching activity on the signals of a circuit, namely [66, 58]:

- the *rising* transition of a signal a, denoted by $a+$, representing the switching from 0 to 1;
- the *falling* transition of a signal a, denoted by $a-$, representing the switching from 1 to 0;
- the *toggle* transition of a signal a, denoted by $a*$, representing that signal a either rises or falls according with its current value;
- the *dummy* transition, denoted by ϵ, used for representing synchronization and some other conditional behaviors (for instance, to non-deterministically allow or inhibit the switching of a signal)

Definition 2.19 (Signal Transition Graph)
An Signal Transition Graph is a 4-tuple $C = (N, M_0, E, \Lambda)$*, where*

- $N = \langle P, T, F \rangle$ *is an ordinary Petri net structure;*
- M_0 *is the initial marking;*
- $E \subseteq V \times \{+, -, *\} \cup \{\epsilon\}$ *is the set of events, being* V *the set of signals;*
- $\Lambda : T \mapsto E$ *is the labeling function which assigns an event to each transition in* T*.*

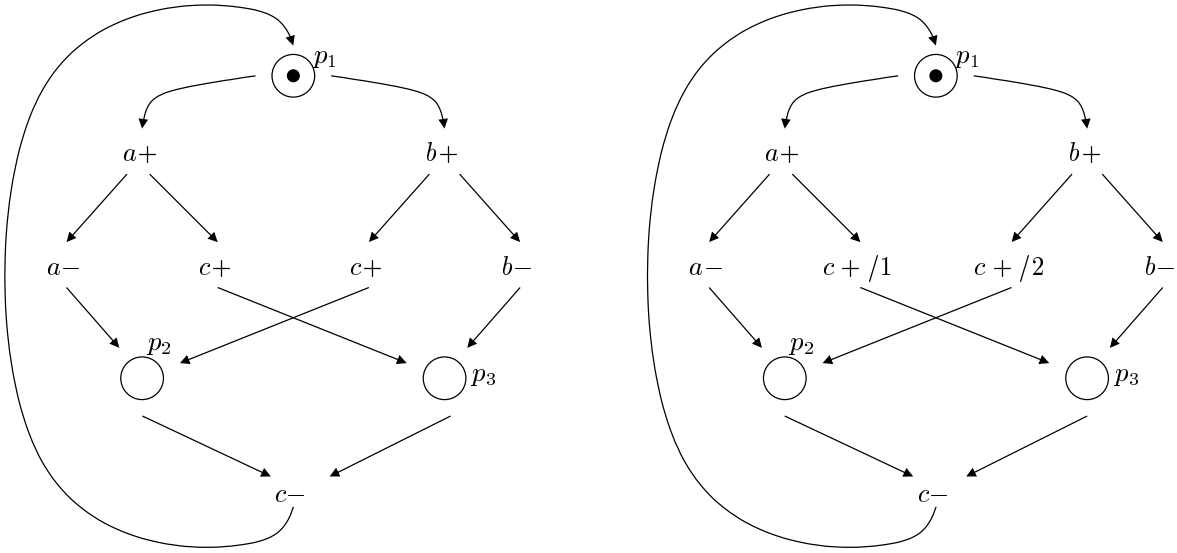


Figure 2.10: (a) A signal transition system (STG). (b) The same STG but with different occurrences of the same event discriminated by means of indices.

The conventional graphical representation for STG is slightly simplified relatively to the usual PN representation. Transitions are represented simply by their label names, places are denoted by circles and elements of the flow relation are represented by directed arcs. Places with only one predecessor and one successor are usually omitted, in which case a directed arc connect the predecessor transition to the successor one. Tokens in these *implicit places* are represented by black dots on the arcs. Figure 2.10.a shows an STG with 6 events, 7 transitions and 7 places, 4 of them implicit. In this figure there are 2 transitions labeled with the same event, $c+$. Representing a transition just by its label makes it difficult to distinguish different occurrences of the same event, especially if we are representing the STG in some textual format. In order to make the distinction a different index is often added to each occurrence of the same event. The STG in figure 2.10.a is repeated in figure 2.10 with the two occurrences of event $c+$ differentiate by means of indices.

In textual representation of STG implicit places are normally defined by the pair of transitions it lies in between. For instance, in figure 2.10, $< a+, a- >$ is the implicit place with $a+$ as the predecessor transition and $a-$ as the successor transition.

As STG can explicitly capture the *causality*, *concurrency* and *choice* relations between signal transitions existent in circuits, it can formally capture the information often represented by timing diagrams. In the initial marking of the STG in figure 2.10, both events $a+$ and $b+$ are enabled, but only one of them can occur. There must exist a *choice* between the two events. If, for instance $a+$ fires, the system evolves to a marking where both events $a-$ and $c+/1$ are enabled. But now these two events can occur *concurrently*.

2.5 Theory of Regions

There is a one to one relation between markings of a Petri net and states of its reachability graph. A given place p of the Petri net will eventually participate in more than one marking. Thus, there must be a set of states r in the reachability graph, correspondent to that place p. Place p eventually has a set of incoming arcs, the input transitions, and a set of outgoing arcs, the output transitions. These transitions represent the transitions of the reachability graph that enter and exit set r. Let us illustrate these ideas with an example.

Consider, for instance, the transition system depicted in figure 2.11, which is the reachability graph of the STG in figure 2.10. The names given to the states represent the corresponding markings in the STG. For instance, the initial state, the one with a double circle, is called p_1, because it corresponds to a marking with one token in place p_1 and zero tokens in the other places. The implicit places $\langle a+, a-\rangle$, $\langle a+, c+/1\rangle$, $\langle b+, c+/2\rangle$ and $\langle b+, b-\rangle$ are referred to as p_4, p_5, p_6 and p_7, respectively. The indices in event $c+$ have been kept for illustration purposes.

Place p_1 has one incoming arc, from transition $c-$, and two outgoing arcs, to transitions $a+$ and $b+$. All $c-$, $a+$ and $b+$ have a single occurrence in the Petri net. Thus, place p_1 corresponds to the set of states $r_1 = \{p_1\}$, entered by $c-$ and exited by $a+$ or $b+$. Note that, in the reachability graph, all transitions labeled with $c-$ enter set r_1, while all transitions labeled with $a+$ or $b+$ exit r_1. All other transitions are either internal or external to the set, thus none crossing it. A set of states where all transitions labeled with the same event have the same "entry/exit" relation is called a *region* [19] and will be formally defined below. Set r_1 is a region.

Place p_2 is a little more interesting. It has two incoming arcs, from transitions $a-$ and $c+/2$, and one outgoing arc, to transition $c-$, and corresponds to the set of states $r_2 = \{p_2p_5, p_2p_3, p_2p_7\}$. If

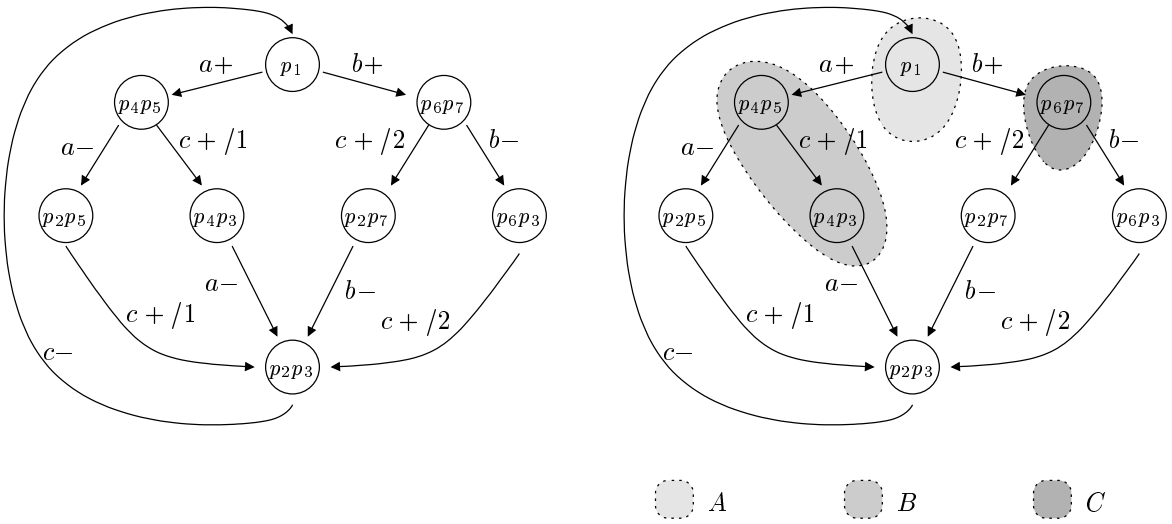


Figure 2.11: (a) The reachability graph for the STG in figure 2.10. (b) the same graph with some regions marked: (A) the region entered by $c-$ and exited by $a+$ or $b+$, (B) the excitation region for event $a-$; (C) the switching region for event $b+$.

we consider $c+/1$ and $c+/2$ as different events, r_2 is a region; otherwise it is not. Indeed, the two transitions labeled with $c+/2$ enter set r_2, while the two labeled with $c+/1$ do not cross r_2. For safe, single event Petri nets there is a one to one correspondence from places in the Petri net to regions in its reachability graph.

There is also a relation between transitions of a Petri net and states of its reachability graph. A Petri net transition is enabled if all its predecessor places are marked. Thus, the set of states correspondent to markings where these places are all marked must have that transitions enabled. Such a set of states is called an *excitation region* [47]. Consider, for instance, in the Petri net of figure 2.10, transition $a-$. Its pre-condition corresponds to place p_4 marked, which in turn corresponds to the set of states $r_3 = \{p_4p_5, p_4p_3\}$ in figure 2.11. Clearly, this is the set of all states where event $a-$ is enabled, that is, the excitation region for event $a-$.

A more interesting situation occurs with event $c+$. If we consider only occurrence $c+/1$, its pre-condition corresponds to place p_5 marked, which in turn corresponds to the set of states $r_4 = \{p_2p_5, p_4p_5\}$. This set has all states with event $c+$ enabled, but there are other states, not included in r_4, with the same event enabled. If we consider only occurrence $c+/2$, the pre-condition corresponds

to place p_6 marked, and the corresponding reachability graph set of states is $r_5 = \{p_6p_7, p_6p_3\}$. This set is composed of all states where $c+$ is enabled, not included in set r_4. As we will understand below, r_4 and r_5 are elementary excitation regions and $r_6 = r_4 \cup r_5$ is a generalized excitation region.

When a transition fires, tokens are added to the successor places of that transition. These marked places represent a set of markings, which in turn represent a set of states in the reachability graph. This set of states is called a *switching* region and represent the set of states reached by the firing of a given event. Consider transition $b+$ in the Petri net of figure 2.10. Its post-condition corresponds to places p_6 and p_7 marked. There is a single state in the reachability graph corresponding to both p_6 and p_7 marked. Thus, the switching region for event $b+$ is the set $r_7 = \{p_6p_7\}$.

Next we are given formal definitions for excitation region, switching region and region. Let $G = \langle S, E, \Theta, s_{in}\rangle$ be a transition system and $e \in E$ an event.

Definition 2.20 (excitation region)

- *A set $r \subseteq S$ is called the* generalized excitation region *for event e if it is the maximal set of states where event e is enabled, that is, the following conditions are true:*

  1. $\forall_{s\in r}, \exists_{s'\in S} : \langle s, e, s'\rangle \in \Theta$
  2. $\forall_{s\notin r}, \forall_{s'\in S} : \langle s, e, s'\rangle \notin \Theta$

- *A set $r \subseteq S$ is called an* (elementary) excitation region *for event e if it is the maximal connected set of states where event e is enabled, that is, the following conditions are true:*

  1. $\forall_{s\in r}, \exists_{s'\in r} : \langle s, e, s'\rangle \in \Theta$
  2. *the transition sub-system induced by r is a connected component.*[4]

The generalized excitation region for event e is the union of all elementary excitation regions for event e.

Definition 2.21 (switching region)

- *A set $r \subseteq S$ is called the* generalized switching region *for event e if it is the maximal set of states reached by an occurrence of event e, that is, the following conditions are true:*

  1. $\forall_{s\in r}, \exists_{s'\in S} : \langle s', e, s\rangle \in \Theta$

[4] For a definition of connected component see section 2.1.4.

2. $\forall_{s \notin r}, \forall_{s' \in S} : \langle s', e, s \rangle \notin \Theta$

- *A set $r \subseteq S$ is called an* (elementary) switching region *for event e if it is the maximal connected set of states reached by an occurrence of e, that is, the following conditions are true:*

  1. $\forall_{s \in r}, \exists_{s' \in S} : \langle s', e, s \rangle \in \Theta$
  2. *the transition sub-system induced by r is a connected component.*

The generalized switching region for event e is the union of all elementary switching regions for event e.

Definition 2.22 (region)

A subset of states $r \subseteq S$ is a region in G if and only if the following condition is satisfied:

$$\begin{aligned}
&\forall_{e \in E}, \forall_{\langle s, e, s' \rangle, \langle x, e, x' \rangle \in \Theta} : \\
&\quad ((s \in r \wedge s' \notin r) \Rightarrow (x \in r \wedge x' \notin r)) \ \vee \\
&\quad ((s \notin r \wedge s' \in r) \Rightarrow (x \notin r \wedge x' \in r)) \ \vee \\
&\quad ((s, s' \in r \vee s, s' \notin r) \Rightarrow (x, x' \in r \vee x, x' \notin r))
\end{aligned}$$

There are two trivial regions in each transition system: the set of all states and the empty set of states.

Definition 2.23 (sub-region)

A region r' is said to be a sub-region *of another region r if and only if $r' \subset r$.*

Definition 2.24 (minimal region)

A non-empty region r is minimal *if there is no non-empty region r' such that $r' \subset r$.*

Some important properties are stated for regions:

Property 2.25

If r and r' are two regions and $r' \subset r$, then $r - r'$ is a region.

Property 2.26

r is a region if and only if $S - r$ is a region, where S is the set of all states.

Property 2.27

Every region can be represented as a union of disjoint minimal regions.

The proof for the previous 3 properties can be found respectively in [9], [62] and [24]. Actually, property 2.26 is a corollary of property 2.25.

The symbol $\bullet$ is used to represent different relations between regions and events. Let $G = \langle S, E, \Theta, s_n \rangle$ be a transition system, r be a region and e an event. Let R be the set of all regions definable on G. Expression $e \bullet r$ means that e is an entry event for region r; similarly, the expression $r \bullet e$ means that e is an exit event for region r;

Definition 2.28 (post-region)

1. *A region r such that $e \bullet r$ is called a* post-region *of e.*
2. *The set of all post-regions of event e is denoted by $e\bullet$, that is, $e\bullet = \{r \in R | e \bullet r\}$.*

Definition 2.29 (pre-region)

1. *A region r such that $r \bullet e$ is called a* pre-region *of e.*
2. *The set of all pre-regions of event e is denoted by $\bullet e$, that is, $\bullet e = \{r \in R | r \bullet e\}$.*

Definition 2.30 (region interface)

1. *The set of events $\bullet r = \{e \in E | e \bullet r\}$ is called the* input interface *of region r.*
2. *The set of events $r\bullet = \{e \in E | r \bullet e\}$ is called the* output interface *of region r.*
3. *The pair $\langle \bullet r, r\bullet \rangle$ is called the* interface *of region r.*

Theorem 2.31

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system and $r_1, r_2 \subseteq S$ to regions on it. If G is connected, the following expression is true:

$$r_1 = r_2 \iff \langle \bullet r_1, r_1 \bullet \rangle = \langle \bullet r_2, r_2 \bullet \rangle$$

Proof

Proof for sufficiency is straightforward. To prove necessity, let assume $r_1 \neq r_2$. Because G is connected, there is a transition $t = \langle s, e, s' \rangle$ such that, t is internal or external to one of the regions and does cross the other. But then event e is in the interface of the latter region, while it is not in the interface of the former. Thus, $\langle \bullet r_1, r_1 \bullet \rangle \neq \langle \bullet r_2, r_2 \bullet \rangle$.

In next chapters we will be interested in transition systems whose states are all reachable from the initial state. Such transition systems are necessarily connected and thus we can use the interface to represent a region.

2.6 Conclusions

The behavior of asynchronous circuits is usually described at two different levels of abstraction. At the state level, an asynchronous circuit is represented by a finite state machine, where evolution from state to state is only commanded by the occurrence of transitions on the signals of the circuit. Transition systems and state graphs are graph formalisms used to represent and manipulate circuit behaviors at state level. State graphs are specific for describing circuits, actually being a specialization of the more general transition system formalism. We will use state graphs to represent circuit behavior and transition systems to represent manipulations to a circuit description.

The event level represents a higher level of abstraction in describing the behavior of an asynchronous circuit. Behavior is put in terms of relations between events. Causality, concurrency and choice between events can be captured explicitly at this level. Petri nets and signal transition graphs are graph formalisms used to represent circuit behavior at event level. Again, signal transition graphs are specific for describing circuit behaviors, actually being a specialization of the more general Petri net formalism.

Most of our work was done at state level, thus making transition system and state graphs the most used formalisms along the thesis. In some situations, however, a state level description appears to be hardly convenient, because it is too large or because it is difficult to emphasize the desired item. In such cases we prefer to use an event level description making things more clear.

Although, as mentioned, we have worked at state level, most of the manipulations done to circuit descriptions are based on events. The notion of region thus plays an important role along the thesis. Regions are sets of states with some fixed event relation. Making a transformation on a state graph based in a region is equivalent to do some kind of event manipulation. Regions will be extensively used in the remaining chapters.

Chapter 3

Transformations at State Level

Along this thesis transition systems (state graphs) are used as the preferred model to represent the behaviour of asynchronous systems. These transition systems will be submitted to different types of transformations in order to make them get some desirable properties. In this chapter we will cover the set of transformations we are interested in. Three different types are considered:

1. Morphisms, a transformation, which maps a transition system into another. A special kind of morphism, called *projection*, will be defined, which "hides" one or more events of a transition system.

2. A composite operation of two or more transition systems, not necessarily state graphs, which take the form of a product.

3. Two basic transformations which represent the insertion of a new event into a transition system and the insertion of a new signal into a state graph. Under certain circunstances these two transformations can be represented in product form.

3.1 Morphisms

Informally a morphism of a transition system into another is a mapping of the states, events and transitions of the former into the states, events and transitions of the latter. Defined in the form of a transition system operation, a morphism transforms a transition system into another with the same

"structure". Seen from a different angle, a morphism describes a way a transition system is simulated by another.

Let $G = \langle S, E, \Theta, s_{in} \rangle$ and $G' = \langle S', E', \Theta', {s'}_{in} \rangle$ be two transition systems.

Definition 3.1 (morphism)

A morphism *from G into G' is a pair $\langle \sigma, \eta \rangle$ where*

- *$\sigma : S \to S'$ is a function which maps the states of G into the states of G', while preserving the initial state, that is, $\sigma(s_{in}) = {s'}_{in}$;*
- *$\eta : \Delta \to E'$, with $\Delta \subseteq E$, is a function which maps the subset Δ of the events of G into the events of G';*
- *the following condition is satisfied*

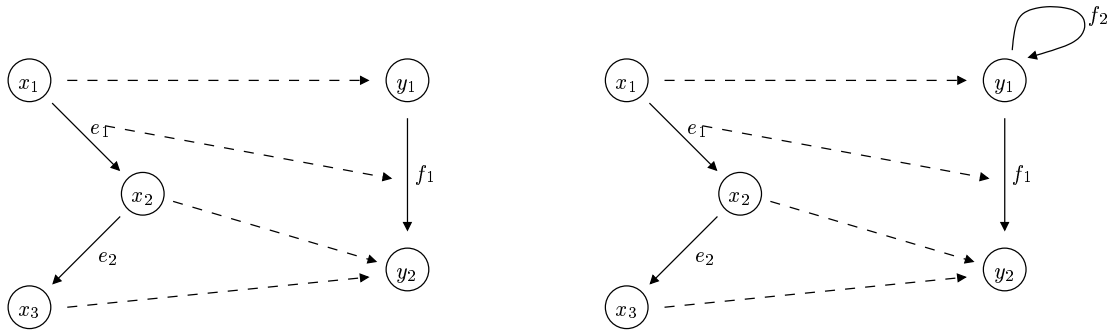
$$\textbf{if} \quad \langle s_u, e, s_v \rangle \in \Theta \quad \textbf{then} \quad \begin{cases} \langle \sigma(s_u), \eta(e), \sigma(s_v) \rangle \in \Theta' & e \in \Delta \\ \sigma(s_u) = \sigma(s_v) & e \notin \Delta \end{cases}$$

Definition 3.1 brings implicitly a mapping on the transition relation. It can be defined as the function $\tau : \Phi \to \Theta'$, where $\Phi \subseteq \Theta$ and $\tau(\langle s_u, e, s_v \rangle) = \langle \sigma(s_u), \eta(e), \sigma(s_v) \rangle$.

Figure 3.1 shows two examples of morphism. For both sets Δ and Φ and functions σ, η and τ are defined as follow:

$$\Delta = \{e_1\}$$
$$\Phi = \{\langle x_1, e_1, x_2 \rangle\}$$
$$\sigma(x_1) = y_1, \quad \sigma(x_2) = \sigma(x_3) = y_2$$
$$\eta(e_1) = f_1$$
$$\tau(\langle x_1, e_1, x_2 \rangle) = \langle y_1, f_1, y_2 \rangle$$

Because η applies to only a subset of the events, a morphism allows for hiding some events of the source transition system. In both examples in figure 3.1 event e_2 is hidden by the morphisms. Definition 3.1 does not impose surjectivity to functions σ or η. This allows for the existence of states, events and transitions in the target transition system images of no state, event and transition in the source. For instance, in the example of figure 3.1.b, functions η and τ are not surjective. Event f_2 and transition $\langle y_1, f_2, y_1 \rangle$ have no correspondence in the source transition system.



(a) (b)

Figure 3.1: Two examples of morphisms. (a) A surjective morphism: every state, event and transition of the transition system on the right side is the image of some state, event or transition on the left. (b) A non-surjective morphism: event f_2 and transition $\langle y_1, f_2, y_1 \rangle$ have no correspondence on the left side.

Different types of morphisms have been defined by different authors; see for instance [3, 2, 1, 25]. Bednardzyk in [2] defines a *rigid morphism* as the one where $E' \subseteq E$ and η is the identity function induced by E', that is, $\eta : \Delta \to E'$, with $\Delta = E'$ and $\eta(e) = e$. Figure 3.2.a shows an example of a rigid morphism.

In [24], Cortadella *et al.* define *split-morphism* as a morphism where σ is a bijection, $\Delta = E$, and η is a surjection. Also it imposes that $\langle s_u, e, s_v \rangle \in \Theta$ if and only if $\langle \sigma(s_u), \eta(e), \sigma(s_v) \rangle \in \Theta'$, which, in conjunction with the bijectivity of σ and the surjectivity of η, imposes τ to be a surjection. More, if we assume no multiple arcs between the same pair of ordered states are allowed, it is also an injection. Figure 3.2.b shows an example of a split-morphism.

Arnold in [1] defines *homomorphism* as a morphism where η is the identity function defined on E, that is, $\Delta = E' = E$ and $\eta : E \to E$, being $\eta(e) = e$. In his definition function τ appears represented explicitly. On the other side, a homomorphism is assumed as a transformation from one transition system to another with the same set of events, and thus function η is not given explicitly. Thus, a homomorphism is defined by the tuple $\langle \sigma, \tau \rangle$. An homomorphism is said to be surjective if σ and η are surjections. If h is a surjective homomorphism from G to G', G' is called the *quotient* of G by h. Figure 3.2.c shows an example of a homomorphism.

If σ, η and τ are bijections the morphism is called an *isomorphism*. Figure 3.2.d shows an example

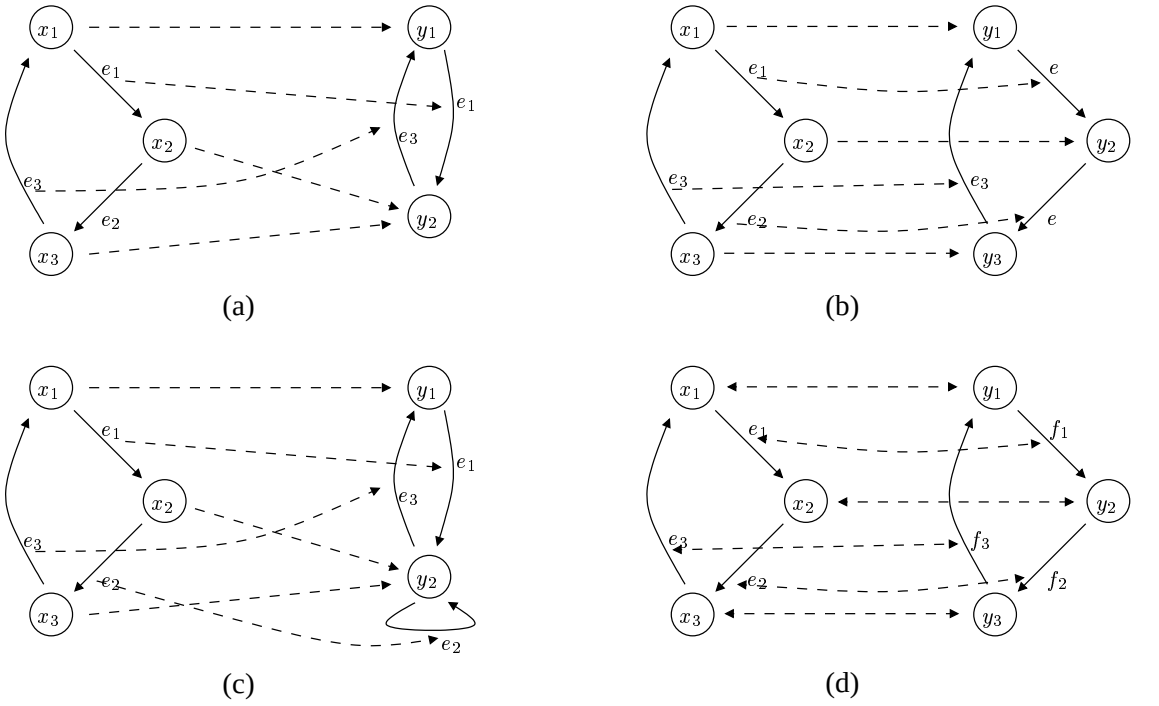


Figure 3.2: Examples of different type of morphisms: (a) rigid morphism; (b) split-morphism; (c) homomorphism; (d) isomorphism.

of an isomorphism.

Definition 3.2 (isomorphism)

An isomorphism *from G to G' is a pair $\langle \sigma, \eta \rangle$ where*

- $\sigma : S \to S'$ *is a bijection on states which preserves initial state, that is,* $\sigma(s_{in}) = {s'}_{in}$;
- $\eta : E \to E'$ *is a bijection on events; and*
- $\langle s_u, e, s_v \rangle \in \Theta \iff \langle \sigma(s_u), \eta(e), \sigma(s_v) \rangle \in \Theta'$

If G is isomorphic to G', G' is isomorphic to G. If G and G' are isomorphic, we denote that fact by $G \cong G'$.

3.2 Projection

Consider a transition system where some of its events are, for instance, *non-observable*. The source and destination states of transitions labeled with these "invisible" events are thus indistinguishable. The idea behind projection is to define a morphism which hides these non-observable events.

Before formally introducing the definitions of projection let us review some needed concepts. A *binary relation* R on a set S is a set of ordered pairs constructed over S, that is, $R \subseteq S \times S$. The assertion $\langle s_u, s_v \rangle \in R$ is usually abbreviated to $s_u R s_v$, and we say s_u *is related to* s_v *by* R. Relation R is *reflexive* if for all $s_u \in S$, $s_u R s_u$; it is *symmetric* if, for all $s_u, s_v \in S$, $s_u R s_v \Rightarrow s_v R s_u$; it is *transitive* if for all $s_u, s_v, s_w \in S$, $(s_u R s_v \wedge s_v R s_w) \Rightarrow s_u R s_w$.

A reflexive, symmetric, and transitive binary relation is called an *equivalence relation*. In an equivalence relation $\sim$ the set of all elements of S that are equivalent to a given element s_u is called the *equivalence class* of s_u, and is denoted by $[s_u]_\sim$, or simply by $[s_u]$ if no confusion arises about the equivalence relation. Thus $[s_u] = \{s \in S \mid s \sim s_u\}$. If $[s_u] \neq [s_v]$, for a given equivalence relation, then $[s_u] \cap [s_v] = \emptyset$. The set of equivalence classes determined by a given equivalence relation $\sim$ on S is denoted by $S/\sim$.

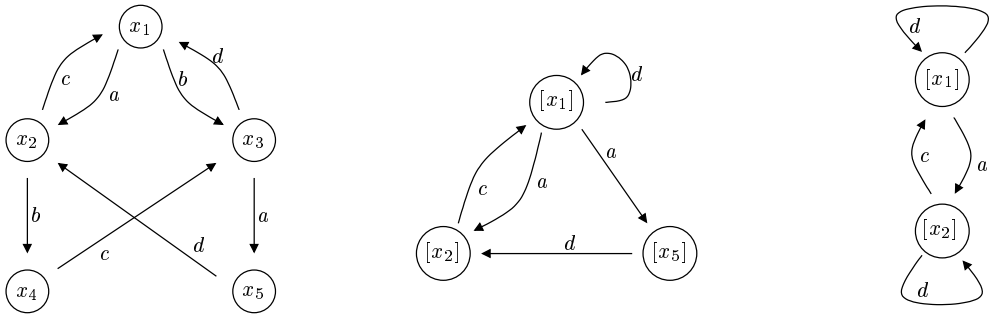
A *partition* of the set S is a division of S into subsets, such that each element of S belongs to exactly one subset. There is a converse correspondence between equivalence classes and partitions on a given set S: if $\sim$ is an equivalence relation on S then $S/\sim$ is a partition of S; conversely, if I is a partition of S then there is an equivalence relation $\sim$ such that $I = S/\sim$.

Definition 3.3 (basic projection)
Let $G = \langle S, E, \Theta, s_{in} \rangle$ *be a transition system and let* $E' \subseteq E$. *The* basic projection *of* G *by* E', *denoted* $G \downarrow E'$, *is the transition system* $G' = \langle S', E', \Theta', {s'}_{in} \rangle$ *where*

$$S' = S/\Delta$$
$$\Theta' = \{\langle [s_u], e, [s_v] \rangle \in S' \times E' \times S' \mid \langle s_u, e, s_v \rangle \in \Theta \quad \textbf{and} \quad e \in E'\}$$
$${s'}_{in} = [s_{in}]$$

Δ *being an equivalence relation on* S *defined inductively by:*

$$\textbf{if} \quad (\langle s_u, e, s_v \rangle \in \Theta \quad \textbf{and} \quad e \notin E') \quad \textbf{then} \quad s_u \Delta s_v$$



(a) (b) (c)

Figure 3.3: Illustrating the concept of projection: (a) original transition system; (b) its basic projection by $\{a, c, d\}$; (c) its deterministic projection by $\{a, c, d\}$.

The previous definition determines both that

$$\textbf{if} \quad \langle s_u, e, s_v \rangle \in \Theta \quad \textbf{then} \quad \begin{cases} \langle [s_u], e, [s_v] \rangle \in \Theta' & e \in E' \\ [s_u] = [s_v] & e \notin E' \end{cases}$$

and that

$$\textbf{if} \quad \langle [s_u], e, [s_v] \rangle \in \Theta' \quad \textbf{then} \quad \langle s_u, e, s_v \rangle \in \Theta$$

So, the basic projection represents a surjective morphism from G into G'.

Figure 3.3.b shows the basic projection of the transition system at its left by the subset of events $\{a, c, d\}$. For it we have $[x_1] = \{x_1, x_3\}$, $[x_2] = \{x_2, x_4\}$ and $[x_5] = \{x_5\}$. Event b is hidden by the projection. This example also shows that the basic projection can transform a deterministic into a non-deterministic transition system. G is deterministic, but G_b is non-deterministic: there are two transitions leaving state $[x_1]$ labeled with the same event a.

Definition 3.4 (deterministic projection)

The deterministic projection *of G by E', denoted by $G \Downarrow E'$, is the transition system $G' = \langle S', E', \Theta', s'_0 \rangle$ where*

$$S' = S/\Delta$$
$$\Theta' = \{\langle [s_u], e, [s_v] \rangle \in S' \times E' \times S' \mid \langle s_u, e, s_v \rangle \in \Theta \quad \textbf{and} \quad e \in E'\}$$
$$s'_0 = [s_0]$$

Δ being an equivalence relation on S defined inductively by:

- **if** $(\langle s_u, e, s_v\rangle \in \Theta$ **and** $e \notin E')$ **then** $s_u \Delta s_v$
- **if** $(s_u \Delta s_v$ **and** $\langle s_u, e, s_w\rangle \in \Theta$ **and** $\langle s_v, e, s_x\rangle \in \Theta)$ **then** $s_w \Delta s_x$

The difference between both definitions is in the equivalence relation: states which cause non determinism in the former are put in the same equivalence class in the latter. As an example, see figure 3.3, where $G_d = G \Downarrow \{a, c, d\}$. In it $[x_1] = \{x_1, x_3\}$, $[x_2] = \{x_2, x_4, x_5\}$. The definition of deterministic projection is quite similar to the definition of *collapse* found for instance in [59] and [2] and applied to concurrent asynchronous systems.[1]

3.2.1 Projection on State Graphs

When talking about state graphs, it makes no sense hiding event e without hiding its complement $\neg e$. Thus any projection on a state graph should be determined by the subset of events induced by a subset of signals. Moreover, since a state graph must be deterministic, basic projection can not be applied. Thus whenever we make projections on state graphs we are referring to a deterministic projection based on a set of events induced by a set of signals. We are fixing this with the following definition

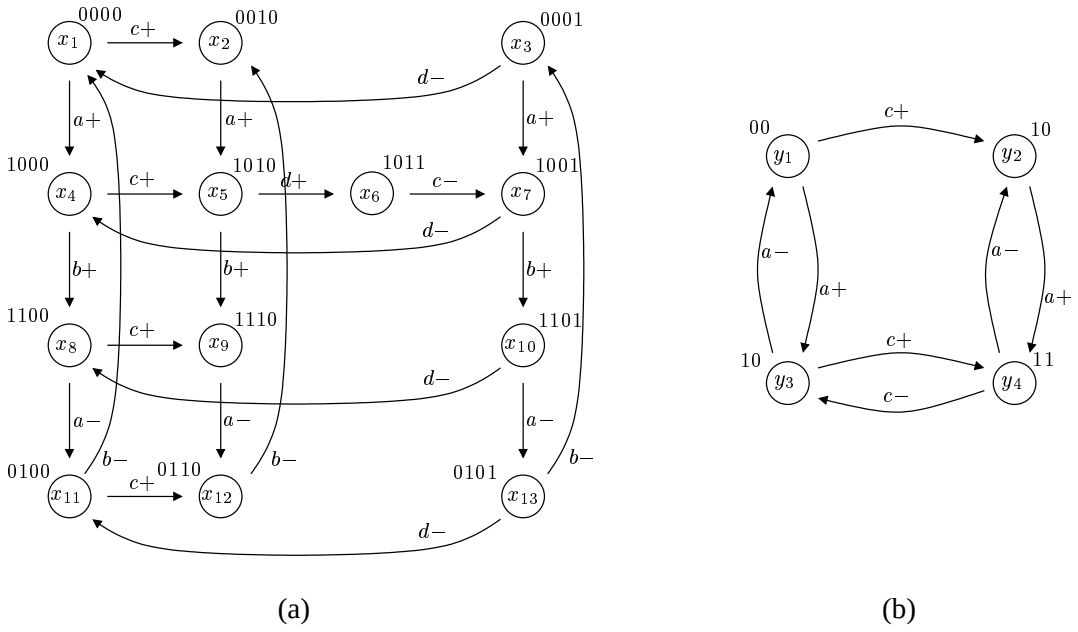
Definition 3.5 (signal projection)
Let $G = \langle S, E, \Theta, s_{in}\rangle$, with $E = V \times \{+, -\}$, be a state graph, where V is the set of signals. Let $V' \subseteq V$ be a subset of signals. The deterministic projection $G \Downarrow (V' \times \{+, -\})$ is called the signal projection of G by V' and is denoted by $G \Downarrow V'$.

Note that the same operator — the double down arrow ($\Downarrow$) — is used to represent both a deterministic projection of transition systems and a signal projection of state graphs. Figure 3.4 shows a state graph with set of signals $\{a, b, c, d\}$ and its signal projection by the subset $\{a, c\}$.

When a state graph is a valid representation of a circuit it must be deterministic and switch-count correct. The signal projection preserves this two properties.

Theorem 3.6 (correctness-preservation)
Let G be a state graph and V' a subset of its signals. If G is deterministic and switch-count correct, then $G' = G \Downarrow V'$ is also deterministic and switch-count correct.

[1] A concurrent asynchronous system is basically a transition system plus an independent relation on events. For a more precise definition see section 3.5 in this chapter.



(a) (b)

Figure 3.4: (a) A state graph with set of signals $\{a, b, c, d\}$. (b) Its signal projection by the subset $\{a, c\}$. For the projection we have: $y_1 = \{x_1, x_3, x_{11}, x_{13}\}$, $y_2 = \{x_2, x_{12}\}$, $y_3 = \{x_4, x_7, x_8, x_{10}\}$ and $y_4 = \{x_5, x_6, x_9\}$.

Proof

A signal projection is a deterministic projection and so G' is deterministic. Remains to be proved switch-count correctness preservation.

Let ω be a semi-walk in G and ω' its image by the projection; ω' can be obtained by contracting all transitions labelled with events not in E', that is, by removing all these transitions and merging their end states. The switch count for any signal not hidden by the projection is thus the same in ω and ω'. Thus every semi-walks in G' image of some semi-walk in G is switch count correct.

By the rules of equivalence class construction, given by definition 3.4, there is no way for two not connected states to go to the same equivalent class. Thus, every semi-walk in G' is the image of at least one semi-walk in G. Hence, every semi-walk in G' is switch-count correct.

By the definition of signal projection two different states go to the same equivalence class if

1. they are connected by a transition on a hidden signal;

2. they are reached by the same signal transition from states belonging to the same equivalence class.

In any case their state codes only differ in bits of hidden signals. Thus, the state labelling function for each signal bit preserved by the projection is the same, and so, the state assignment function for the projection can be obtained from the original by deleting bits hidden by the projection. Take figure 3.4 as an example. By the projection states x_1, x_3, x_{11} and x_{13} go to the same equivalence class, which corresponds to state y_1. Bit for signal a has the same value in the state codes of all these five states. The same happens with bit for signal c.

3.3 Product of Transition Systems

Labeled transition systems have been used to model the behavior of systems. When two or more systems work together, we need to correlate the dynamics of the composite system with the dynamics of its components. Two scenarios are possible. In a pure synchronous scenario, the occurrence of an event in one component is necessarily accompanied by the simultaneous occurrence of an event in each one of the other components. A global event, that is, an event in the composite system is defined as a vector of events, one per system component. In a pure asynchronous scenario, an event can occur in one component while no event occurs in some of the others. Assuming the existence of a dummy event associated to every state, which makes the system to move from a state into itself, the asynchronous scenario can be seen as the synchronous one.

The behavior of the composite system can be constrained by reducing the set of acceptable global events. We will assume an asynchronous scenario constrained as follows:

1. events with the same name in different system components are assumed to be the same event;

2. events occur only one at a time;

The operation of product of transition systems ([1]) gives the formalism to represent the behavior of the composite system in terms of the transition system descriptions of the components. As a result of the product we get a transition system describing the overall behavior.

3.3.1 Free product

Definition 3.7 (free product)

Let $G_i = \langle S_i, E_i, \Theta_i, (s_i)_{in} \rangle$, for $i = 1, 2, \cdots, n$, represent n transition systems. The free product *$G_1 \times G_2 \times \cdots \times G_n$ of these transition systems is another transition system $G = \langle S, E, \Theta, s_{in} \rangle$ where*

$$S = S_1 \times S_2 \times \cdots \times S_n$$
$$E = E_1 \times E_2 \times \cdots \times E_n$$
$$\Theta = \Theta_1 \times \Theta_2 \times \cdots \times \Theta_n$$
$$s_{in} = \langle (s_1)_{in}, (s_2)_{in}, \cdots, (s_n)_{in} \rangle$$

This definition makes two assumptions:

1. It assumes the occurrence of events in different systems is synchronous, that is, when occurs an event in one product member, synchronously occurs an event in each one of all the other product members. It is exactly what "says" $E = E_1 \times E_2 \times \cdots \times E_n$ which represents the actions of the composite system G. If $e = \langle e_1, e_2, \cdots, e_n \rangle$ is an element of E, the occurrence of e in G represents the simultaneous occurrence of e_1 in G_1, e_2 in G_2, $\cdots$, and e_n in G_n.

2. There is no interdependency between events in different systems, that is, the ocurrence of one given event in one system does not condition the ocurrence of any event in the others.

Consider, for instance, transition systems G_u and G_v defined as follow, and shown in figure 3.5.a.

$G_u = \langle S_u, E_u, \Theta_u, u_{in} \rangle$
where
$S_u = \{u_1, u_2\}$
$E_u = \{e_1, e_2\}$
$\Theta_u = \{\langle u_1, e_1, u_2 \rangle, \langle u_2, e_2, u_2 \rangle\}$
$u_{in} = u_1$

$G_v = \langle S_v, E_v, \Theta_v, v_{in} \rangle$
where
$S_v = \{v_1, v_2\}$
$E_v = \{f_1, f_2\}$
$\Theta_v = \{\langle v_1, f_1, v_1 \rangle, \langle v_1, f_2, v_2 \rangle\}$
$v_{in} = v_1$

The transition system G, shown in figure 3.5.b, is the result of the free product between G_u and G_v ($G = G_u \times G_v$). It has 4 states, 4 events and 4 transitions. One of the states is not reachable from the

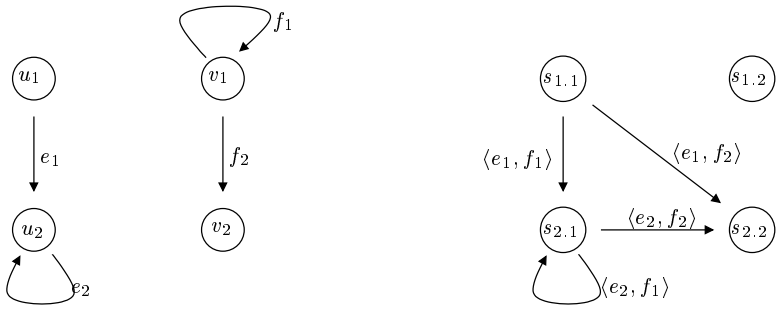


Figure 3.5: Illustrating the free product between transition systems: (a) product operands; (b) product result. Global state $s_{i.j}$ represents the ordered pair $\langle u_i, v_j \rangle$.

initial state. Actually $G = \langle S, T, \Theta, s_{in} \rangle$, where

$$
\begin{aligned}
S &= S_u \times S_v = \{s_{1.1}, s_{1.2}, s_{2.1}, s_{2.2}\} \\
E &= E_u \times E_v = \{\langle e_1, f_1 \rangle, \langle e_1, f_2 \rangle, \langle e_2, f_1 \rangle, \langle e_2, f_2 \rangle\} \\
\Theta &= \Theta_u \times \Theta_v = \{\langle s_{1.1}, \langle e_1, f_1 \rangle, s_{2.1} \rangle, \langle s_{1.1}, \langle e_1, f_2 \rangle, s_{2.2} \rangle, \\
&\quad \langle s_{2.1}, \langle e_2, f_1 \rangle, s_{2.1} \rangle, \langle s_{2.1}, \langle e_2, f_2 \rangle, s_{2.2} \rangle\} \\
s_{in} &= \langle u_{in}, v_{in} \rangle = \{s_{1.1}\}
\end{aligned}
$$

and where global state $s_{i.j}$ represents $\langle u_i, v_j \rangle$.

3.3.2 Constraint Product

Often events in the composite system are interdependent. The ocurrence of an event in one system conditions the ocurrence of some event in another system. Consider for instance that event e_u of system G_u is in mutual exclusion with event e_v of system G_v. Thus no global event can contain both event e_u for G_u and e_v for G_v. As another example consider that events e_u and e_v are synchronous: when the former occurs in G_u, the latter also occur in G_v. In this case the global event set can not contain elements with e_u (e_v) and without e_v (e_u). A similar situation occurs when there are events common to two or more systems.

Keeping the synchronous assumption but considering events in different systems can be interdependent, some of the events in $E_1 \times E_2 \times \cdots \times E_n$ cannot occur in the composite system. The set of

events of the composite system is thus a subset of this Cartesian product.

Definition 3.8 (constraint)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system and $I \subseteq E$. The constraint *of G by I, denoted by $G[I]$, is the transition subsystem obtained from G by removing all events not in I along with all associated transitions.*

Arnold ([1]) defines the *synchronous product* of $G_1, G_2, \cdots, G_n$ in relation to constraint I, denoted by $\langle G_1, G_2, \cdots, G_n; I \rangle$, as the transition subsystem of the free product $G_1 \times G_2 \times \cdots \times G_n$ containing only transitions labeled with an element of I. His synchronous product is thus equivalent to the constraint of G by I, being G the free product of the n G_i. Arnold calls *synchronization constraint* the set I, and *synchronization vector* each one of its elements.

3.3.3 "Asynchronous" Free Product

Let now relax the synchronous assumption and consider that events in different systems are time unrelated, except for specific constraints added to the behavior. Consider for instance that the systems in figure 3.5.a, repeated in figure 3.6.a, are time unrelated one from the other. Then, the expected behavior for the composite system is not the one in figure 3.5.b, but instead the one in figure 3.6.b. Here events in the individual systems can occur independently from each other, as well as in simultaneity.

This "asynchronous" free product can be obtained via the free product by making some pre- and post-processing. In the pre-processing phase we "extend" the transition system description of each individual system by:

- adding a *dummy event* to its set of events;
- adding a self-loop transition labeled with the dummy event to every state.

In the post-processing phase we remove the dummy event of the composite system.

Definition 3.9 (dummy extension)

Let ε_i be the dummy event of transition system G_i. The dummy-extension *of G_i is the transition system $G_i^* = \langle S_i, E_i^*, \Theta_i^*, s_{in} \rangle$ where*

$$E_i^* = E_i \cup \{\varepsilon_i\}$$
$$\Theta_i^* = \Theta_i \cup \{\langle s_i, \varepsilon_i, s_i \rangle \mid s_i \in S_i\}$$

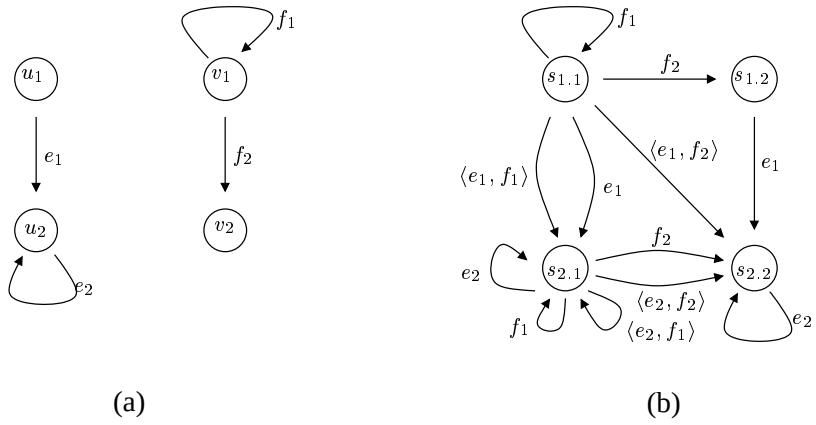


Figure 3.6: Illustrating the asynchronous free product between transition systems: (a) product operands; (b) product result. Global state $s_{i.j}$ represents the pair $\langle u_i, v_j \rangle$.

The set of events of the free product of the dummy-extensions G_i^* is given by

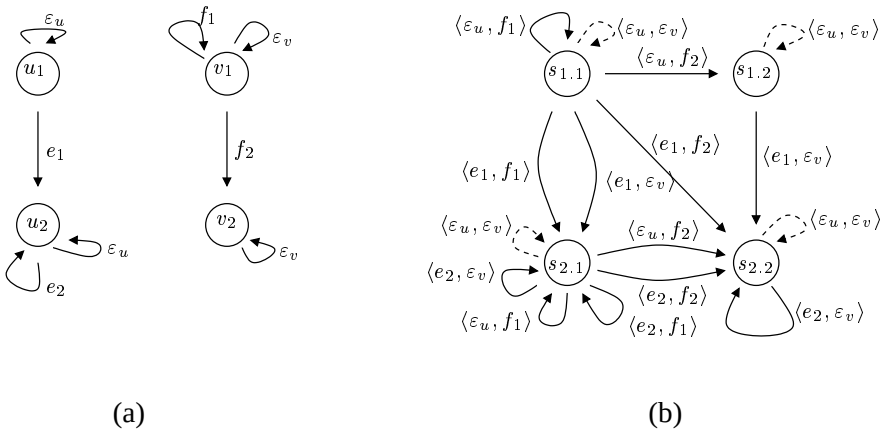
$$E^{\varepsilon} = (E_1 \cup \{\varepsilon_1\}) \times (E_2 \cup \{\varepsilon_2\}) \times \cdots \times (E_n \cup \{\varepsilon_n\})$$

In it we can find elements corresponding to a non-dummy event in one of the members and dummy events in all the others. This means the occurrence of an event in one of the individual system makes the composite system change state. Also all combinations from 2 to n non-dummies can be found. Thus all kind of simultaneity between events of the individual systems are possible. The global dummy event, $\langle \varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n \rangle$, which corresponds to the occurrence of dummy events in all the individual systems, is also an element of the global event set. In the post-product processing phase we just remove it from the composite transition system along with all its associated transitions, self-loops in the global states.

Definition 3.10 (asynchronous free product)

Let $G_i = \langle S_i, E_i, \Theta_i, (s_i)_{in} \rangle$, for $i = 1, 2, \cdots, n$, represent n transition systems. The asynchronous free product *of these transition systems, also denoted by $G_1 \times G_2 \times \cdots \times G_n$, is another transition system given by the constraint $G^*[I]$, where G^* is the free product of the dummy-extensions of all G_i and*

$$I = E^* - \{\langle \varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n \rangle\}$$



(a) (b)

Figure 3.7: Illustrating the use of dummy events to obtain the asynchronous free product: (a) Dummy-extensions of the product operands in figure 3.6.a. (b) Free product result. Global state $s_{i.j}$ represents the pair $\langle u_i, v_j \rangle$. Removing the global dummy event $\langle \varepsilon_u, \varepsilon_v \rangle$ and all associated transitions, this transition system becomes the one in figure 3.6.b.

is the synchronization constraint which removes the global dummy event.

Let illustrate this with the example in figure 3.6. The dummy-extension of G_u and G_v are

$G_u^* = \langle S_u, E_u^*, \Theta_u^*, u_{in} \rangle$
where
$S_u = \{u_1, u_2\}$
$E_u^* = \{e_1, e_2, \varepsilon_u\}$
$\Theta_u^* = \{\langle u_1, e_1, u_2 \rangle, \langle u_2, e_2, u_2 \rangle,$
$\langle u_1, \varepsilon_u, u_1 \rangle, \langle u_2, \varepsilon_u, u_2 \rangle\}$
$u_{in} = u_1$

$G_v^* = \langle S_v, E_v^*, \Theta_v^*, v_{in} \rangle$
where
$S_v = \{v_1, v_2\}$
$E_v^* = \{f_1, f_2, \varepsilon_v\}$
$\Theta_v^* = \{\langle v_1, f_1, v_1 \rangle, \langle v_1, f_2, v_2 \rangle,$
$\langle v_1, \varepsilon_v, v_1 \rangle, \langle v_2, \varepsilon_v, v_2 \rangle\}$
$v_{in} = v_1$

and are depicted in figure 3.7.a. The free product $G^* = G_u^* \times G_v^*$ is the transition system depicted in figure 3.7.b. If we remove from G^* the global dummy event $\langle \varepsilon_u, \varepsilon_v \rangle$ and all associated transitions, we get the expected transition system depicted in figure 3.6.b.

3.3.4 Asynchronous Product

Let now finally formally introduce the product we are interested in. The asynchronous product of transition systems can be generally defined conditioning the asynchronous free product by some synchronization constraint. We are interested in an asynchronous product definition which takes into account the following points:

- Events in different transition systems with the same *name* are actually the same event and thus are necessarily synchronized.
- Different events, i.e., events with different names, can never occur simultaneously.[2]

The set of events of the composite system can thus be defined as the union of the sets of events of the individual ones. This definition actually corresponds to the product used by Bednarczyk ([3, 32]).

Definition 3.11 (asynchronous product)
Let $G_i = \langle S_i, E_i, \Theta_i, (s_i)_{in} \rangle$*, for* $i = 1, 2, \cdots, n$*, represent* n *transition systems. Their* asynchronous product*, denoted by* $G_1 \times G_2 \times \cdots \times G_n$*, is another transition system* $G = \langle S, E, \Theta, s_{in} \rangle$ *where*

$$S = S_1 \times S_2 \times \cdots \times S_n$$

$$E = E_1 \cup E_2 \cup \cdots \cup E_n$$

$$\Theta = \{ \langle u, e, v \rangle \in S \times E \times S \mid \textit{for} \quad i = 1, 2, \cdots, n \\ (e \in E_i \ \wedge \ \langle u_i, e, v_i \rangle \in \Theta_i) \ \vee \ (e \notin E_i \ \wedge \ u_i = v_i) \ \}$$

$$s_{in} = \langle (s_1)_{in}, (s_2)_{in}, \cdots, (s_n)_{in} \rangle$$

Figure 3.8.b shows the transition system result of the asynchronous product of the two transition systems at the left side. The initial states are marked with a double circle. Global state $s_{i,j}$ is equivalent to $\langle u_i, v_j \rangle$. When we are working with initialized transition systems we are not interested in states not reachable from the initial state. Eventually, the transition system, result of an asynchronous product, has states not reachable from the initial state. Such states can be removed without changing the behavior of the composite system. We will denote by **reach**(G) the transition subsystem of G obtained

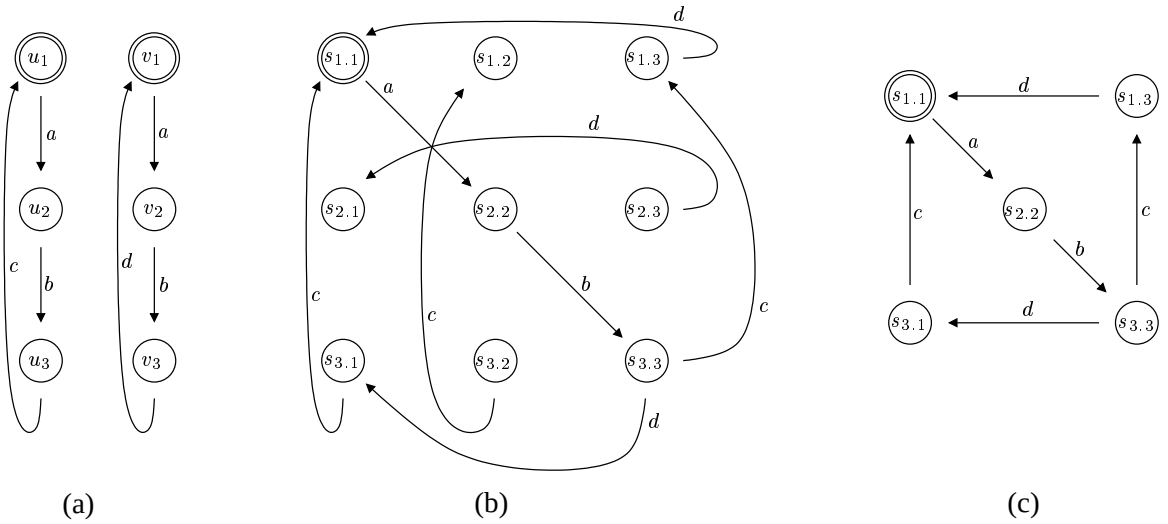


Figure 3.8: Illustrating the asynchronous product of transition systems: (a) product operands; (b) product result; (c) reachable part of the product.

after removing all states not reachable from the initial state. Figure 3.8.c shows the transition system in figure 3.8.b after removing all unreachable states.

A number of usefull properties can be proved to be satisfied by the asynchronous product of transition systems. Some of them are put in terms of isomorphisms between transition systems.

Property 3.12 (determinism)

The asynchronous product of deterministic transition systems is a deterministic transition system.

Proof

Assume the product is not deterministic and let Θ be its transition relation. Then there are two transitions such that $\langle u, e, v\rangle, \langle u, e, w\rangle \in \Theta$. By the definition of asynchronous product, if $e \in E_i$ then $\langle u_i, e, v_i\rangle, \langle u_i, e, w_i\rangle \in \Theta_i$, which contradicts the determinism of the component.

Property 3.13 (associativity)

The asynchronous product of deterministic transition systems is associative. Formally, if G_1, G_2, and G_3 are deterministic transition systems, then $(G_1 \times G_2) \times G_3 = G_1 \times (G_2 \times G_3)$.

[2]State graph definition assumes transitions are labeled with a single signal transition. Thus any valid global event must contain exactly one event different from the dummy events.

Proof

In terms of states and events the proof is straightforward since the Cartesian product and union are associative operations. Let Θ_{12} *and* $\Theta_{(12)3}$ *represent the transition relations for* $G_1 \times G_2$ *and* $(G_1 \times G_2) \times G_3$ *respectively. From the definition of asynchronous product,* $\langle\langle u_1, u_2\rangle, e, \langle v_1, v_2\rangle\rangle \in \Theta_{12}$ *if and only if the two following conditions are truth:*

$$\begin{aligned}
&(e \in E_1 \ \wedge \ \langle u_1, e, v_1\rangle \in \Theta_1) \ \vee \ (e \notin E_1 \ \wedge \ u_1 = v_1) \\
&(e \in E_2 \ \wedge \ \langle u_2, e, v_2\rangle \in \Theta_2) \ \vee \ (e \notin E_2 \ \wedge \ u_2 = v_2)
\end{aligned} \tag{3.1}$$

Also, $\langle\langle u_1, u_2, u_3\rangle, e, \langle v_1, v_2, v_3\rangle\rangle \in \Theta_{(12)3}$ *if and only if the two following conditions are truth:*

$$\begin{aligned}
&(e \in E_{12} \ \wedge \ \overbrace{\langle\langle u_1, u_2\rangle, e, \langle v_1, v_2\rangle\rangle \in \Theta_{12}}^{X_{12}}) \ \vee \ (e \notin E_{12} \ \wedge \ \langle u_1, u_2\rangle = \langle v_1, v_2\rangle) \\
&(e \in E_3 \ \wedge \ \langle u_3, e, v_3\rangle \in \Theta_3) \ \vee \ (e \notin E_3 \ \wedge \ u_3 = v_3)
\end{aligned} \tag{3.2}$$

Replacing expression X_{12} *with conditions 3.1 and working the expression we get*

$$\begin{aligned}
&(e \in E_1 \ \wedge \ \langle u_1, e, v_1\rangle \in \Theta_1) \ \vee \ (e \notin E_1 \ \wedge \ u_1 = v_1) \\
&(e \in E_2 \ \wedge \ \langle u_2, e, v_2\rangle \in \Theta_2) \ \vee \ (e \notin E_2 \ \wedge \ u_2 = v_2) \\
&(e \in E_3 \ \wedge \ \langle u_3, e, v_3\rangle \in \Theta_3) \ \vee \ (e \notin E_3 \ \wedge \ u_3 = v_3)
\end{aligned} \tag{3.3}$$

Working on Θ_{23} *and* $\Theta_{1(23)}$*, the transition relations of* $G_2 \times G_3$ *and* $G_1 \times (G_2 \times G_3)$*, we will get exactly the same conditions. So, the asynchronous product of transition systems is associative.*

Property 3.14 (commutativity)

Up to isomorphism, the asynchronous product of deterministic transition systems is commutative. Formally, if G_1 *and* G_2 *are two deterministic transition systems, then* $G_1 \times G_2 \cong G_2 \times G_1$*.*

Proof

Let $G_{12} = G_1 \times G_2$ *and* $G_{21} = G_2 \times G_1$*. Clearly,* $E_{12} = E_{21}$ *and then function* $\eta : E_{12} \mapsto E_{21}$ *can be defined as the identity function. Let* $\sigma : S_{12} \to S_{21}$ *be a function defined as* $\sigma(\langle u_1, u_2\rangle) = \langle u_2, u_1\rangle$*. Function* σ *is clearly a bijection. Let* $\tau = \Theta_{12} \mapsto \Theta_{21}$ *be a function defined as* $\tau(\langle\langle u_1, u_2\rangle, e, \langle v_1, v_2\rangle\rangle) = \langle\sigma(\langle u_1, u_2\rangle), \eta(e), \sigma(\langle v_1, v_2\rangle)\rangle = \langle\langle u_2, u_1\rangle, e, \langle v_2, v_1\rangle\rangle$*.*

Function τ is injective. Computing the image of Θ_{12} by τ we get

$$\begin{aligned}\tau(\Theta_{12}) &= \tau(\{\langle\langle u_1,u_2\rangle,e,\langle v_1,v_2\rangle\rangle| \\ &\quad ((e\in E_1\wedge\langle u_1,e,v_1\rangle\in\Theta_1)\vee(e\notin E_1\wedge u_1=v_1)\wedge \\ &\quad (e\in E_2\wedge\langle u_2,e,v_2\rangle\in\Theta_2)\vee(e\notin E_2\wedge u_2=v_2))\}) \\ &= \{\langle\langle u_2,u_1\rangle,e,\langle v_2,v_1\rangle\rangle| \\ &\quad ((e\in E_1\wedge\langle u_1,e,v_1\rangle\in\Theta_1)\vee(e\notin E_1\wedge u_1=v_1)\wedge \\ &\quad (e\in E_2\wedge\langle u_2,e,v_2\rangle\in\Theta_2)\vee(e\notin E_2\wedge u_2=v_2))\} \\ &= \Theta_{21}\end{aligned}$$

Thus, function τ is also surjective and hence homomorphism $\langle\sigma,\tau\rangle$ is an isomorphism from G_{12} into G_{21}.

Statement 3.15 (idempotency)

Up to isomorphism and reachability from the initial state, the asynchronous product of deterministic transition systems is idempotent. Formally, if G is a deterministic transition system, then **reach**$(G)\cong$ **reach** $(G\times G)$.

Proof

Let $G\times G = G_X = \langle S_X,E_X,\Theta_X,s_{Xin}\rangle$. For the events, we have $E_X = E\cup E = E$. Moreover, all events are obviously common to both product operands. For states, we have $S_X = S\times S$. Let split this set into two, S'_X and S''_X, being $S'_X = \{\langle u,v\rangle\in S_X \mid u = v$ and $S''_X = \{\langle u,v\rangle\in S_X \mid u <> v$. From the definition of asynchronous product, if $\langle u_1,e,v_1\rangle\in\Theta$ and $\langle u_2,e,v_2\rangle\in\Theta$, then $\langle\langle u_1,u_2\rangle,e,\langle v_1,v_2\rangle\rangle\in\Theta_X$. If $u_1 = u_2$, then v_1 and v_2 are necessarily equal, because we have assumed G is deterministic. Thus, for every transition of Θ_X, if the source state belongs to S'_X, the destination state also does. Since the initial state, $s_{Xin} = \langle s_{in},s_{in}\rangle$, belongs to S'_X, no state from S''_X is reachable from the initial state.

Let $G'_X = \langle S'_X,E_X,\Theta'_X,s_{Xin}\rangle$, be the transition subsystem obtained after deletion of all states from S''_X along with all associated transitions. Thus, $\Theta'_X = \{\langle\langle u,u\rangle,e,\langle v,v\rangle\rangle \mid \langle u,e,v\rangle\in\Theta\}$. Function $\sigma: S\to S'_X$, with $\sigma(u) = \langle u,u\rangle$, function $\eta: E\to E_X$, with $\eta(e) = e$, and function $\tau:\Theta\to\Theta'_X$, with $\tau(\langle u,e,v\rangle) = \langle\langle u,u\rangle,e,\langle v,v\rangle\rangle$ are bijections, thus $G\cong G'_X$ and hence

$$\textbf{reach}(G)\cong\textbf{reach}(G'_X) = \textbf{reach}(G\times G)$$

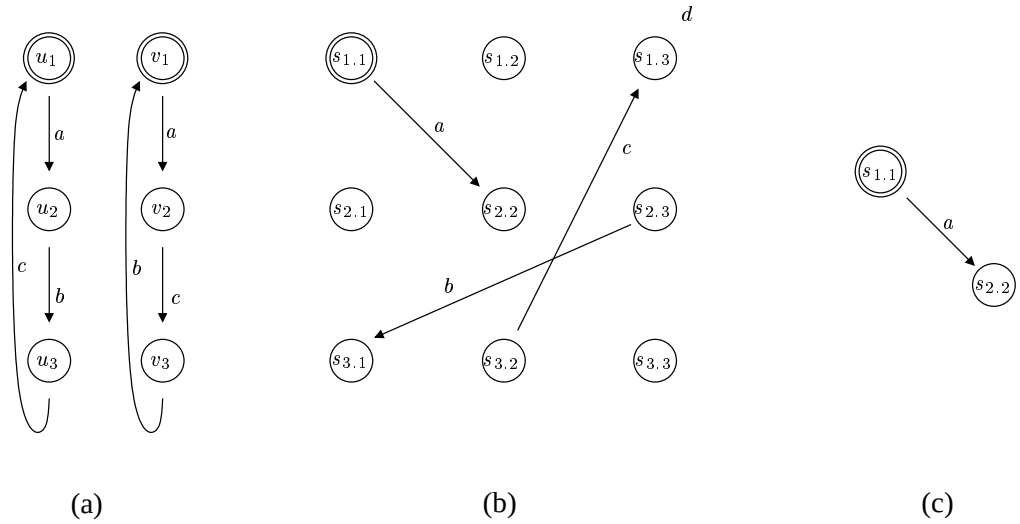


Figure 3.9: An asynchronous product of transition systems with unfeasible events: (a) operands; (b) asynchronous product, where state $s_{i.j}$ represents state $\langle u_i, v_j \rangle$; (c) reachable part of the product.

An event e is said to be *feasible* if there is a state, reachable from the initial state, where e is enabled. Otherwise it is said to be unfeasible. The asynchronous product operation does not preserve feasibility of events. See for instance figure 3.9. Events b and c are feasible in the product operands, but they are unfeasible in the product result.

3.3.5 Product of Projections

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a deterministic transition system. Let $E_1, E_2 \subseteq E$ such that $E_1 \cup E_2 = E$ and consider G_1 and G_2 as the deterministic projections of G by E_1 and E_2, that is, $G_1 = G \Downarrow E_1 = \langle S_1, E_1, \Theta_1, (s_1)_{in} \rangle$ and $G_2 = G \Downarrow E_2 = \langle S_2, E_2, \Theta_2, (s_2)_{in} \rangle$. Is there any relation between G and $G_{12} = G_1 \times G_2$? Or, put in other words, can a homomorphism from G into G_{12} be defined?

Consider $\sim_1$ and $\sim_2$ as the equivalence relations associated respectively to G_1 and G_2. $S_1 = S/\sim_1$ and $S_2 = S/\sim_2$ are partitions of S. Each element $s \in S$ belongs to one and only one element of S_1, which is $[s]_{\sim_1}$. Similarly s belongs to one and only one element of S_2. $S_{12} = S_1 \times S_2$ and so its elements have the form $\langle u, v \rangle$. For each element $s \in S$ there is one and only one element $\langle u, v \rangle \in S_{12}$ such that $u = [s]_{\sim_1}$ and $v = [s]_{\sim_2}$. Thus we can define a function $\sigma_{12} : S \to S_{12}$ which maps each state of S into a state of S_{12}. Function σ_{12} preserves initial state. Indeed, $\sigma_{12}(s_{in}) =$

$\langle [s_{in}]_{\sim_1}, [s_{in}]_{\sim_2} \rangle$ which is the initial state of G_{12}.

Let $\langle s_a, e, s_b \rangle \in \Theta$. By the definition of deterministic projection, if $e \in E_i$, for $i \in \{1,2\}$, then $\langle [s_a]_{\sim_i}, e, [s_b]_{\sim_i} \rangle \in \Theta_i$, otherwise $[s_a]_{\sim_1} = [s_b]_{\sim_i}$. Thus by the definition of product and since $e \in E_1 \cap E_2$, $\langle \langle [s_a]_{\sim_1}, [s_a]_{\sim_2} \rangle, e, \langle [s_b]_{\sim_1}, [s_b]_{\sim_2} \rangle \rangle \in \Theta_{12}$. This transition is equal to $\langle \sigma_{12}(s_a), e, \sigma_{12}(s_b) \rangle$ and so, there is an homomorphism from G into G_{12}. If σ_{12} is injective, G is isomorphic to a transition subsystem of G_{12}, the transition subsystem induced by the set of states $\sigma_{12}(S)$. In this case the interception of a state $u \in S_1$ and a state $v \in S_2$ represents either a single element of S or no element at all. If σ_{12} is bijective, G is isomorphic to G_{12}, and $G_1 \times G_2$ is a factorization of G (see section 3.5). The following theorem expresses the existence of this homomorphism.

Theorem 3.16

Let G be a transition system, and E_1 and E_2 two subsets of its events such that $E_1 \cup E_2$ is equal to the whole set of events of G. There is an homomorphism from G into $(G \Downarrow E_1) \times (G \Downarrow E_2)$.

A more general theorem can also be formulated

Theorem 3.17

Let G be a transition system, and E_1 and E_2 two subsets of its events such that $E_1 \cup E_2$ is equal to the whole set of events of G; let $G_i = G \Downarrow E_i$, for $i = \{1,2\}$; let G'_i, for $i = \{1,2\}$, be two state graphs such that there is an homomorphism from G_i to G'_i; finally, let $G'_{12} = G'_1 \times G'_2$. There is an homomorphism from G into G'_{12}.

Proof

Let, for $i = \{1,2\}$, $\sim_i$ be the equivalent relation associated to projections G_i; let, for $i = \{1,2\}$, $\langle \sigma'_i, \eta'_i \rangle$ be the homomorphism from G_i to G'_i. (Note that η'_i is the identity function on set E_i.) If we follow a similar reasoning as the one done for the previous statement we get the function $\sigma'_x x : S \to S'_{12}$, which map states of G into states of G'_{12}, defined by

$$\sigma'_{12}(s) = \langle \sigma'_1([s]_{\sim_1}), \sigma'_2([s]_{\sim_2}) \rangle$$

In the other side, for each transition $\langle s_a, e, s_b \rangle$ of G there is a transition of G'_{12} defined by

$$\langle \langle \sigma'_1([s_a]_{\sim_1}), \sigma'_2([s_a]_{\sim_2}) \rangle, e, \langle \sigma'_1([s_b]_{\sim_1}), \sigma'_2([s_b]_{\sim_2}) \rangle$$

This transition is equal to $\langle \sigma'_{12}(s_a), e, \sigma'_{12}(s_b) \rangle$ and so, there is an homomorphism from G into G'_{12}.

Theorem 3.16 can also be generalized by augmenting the number of factors.

Theorem 3.18 (homomorphism)
Let G be a transition system; let $E_1, E_2, \cdots, E_n \subseteq E$ be subsets of events such that $E_1 \cup E_2 \cup \cdots \cup E_n = E$; let $G_i = G \Downarrow E_i$, for $i = 1, 2, \cdots, n$: there is an homomorphism from **reach**(G) *into* **reach**$(G_1 \times G_2 \times \cdots \times G_n)$.

3.4 Product of State Graphs

A state graph is a specialized form of transition system. The product of state graphs is thus a product of transition systems with some specificities. Namely

- State graphs represent the behavior of asynchronous circuits and thus the product is asynchronous in nature.

- State graph events are bound to single signal transitions, which are mutual exclusive in nature: under the state graph model signal transitions are assumed to be time separated. Thus all synchronization vectors of the transition system product which represent more than one signal transition must be deleted by the synchronization constraint.

- If a signal is common to two or more circuits, the occurrence of transitions on such signal must occur simultaneously in all circuits. Thus events in different circuits bound to the same signal transition must be synchronized.

- The product of state graphs must be a state graph.

The asynchronous product of transition systems given by definition 3.11 clearly observe three of the previous points. Just the last needs to be analyzed. Let $\{C_i = \langle S_i, V_i \times \{+, -\}, \Theta_i, (s_i)_{in} \rangle \mid i = 1, 2, \cdots, n\}$ be a family of state graphs. Let $C = \langle S, E, \Theta, s_{in} \rangle$ be their asynchronous product as

given by definition 3.11. Thus,

$$
\begin{aligned}
S &= S_1 \times S_2 \times \cdots \times S_n \\
E &= (V_1 \times \{+,-\}) \cup (V_2 \times \{+,-\}) \cup \cdots \cup (V_n \times \{+,-\}) \\
&\quad = (V_1 \cup V_2 \cup \cdots \cup V_n) \times \{+,-\} = V \times \{+,-\} \\
\Theta &= \{\langle s_u, e, s_v\rangle \mid \text{for} i = 1,2,\cdots,n \\
&\quad (e \in V_i \times \{+,-\}) \wedge \langle (s_u)_i, e, (s_v)_i\rangle \in \Theta_i) \ \ \vee \ \ (e \notin V_i \times \{+,-\}) \wedge (s_u)_i = (s_v)_i \\
s_{in} &= \langle (s_1)_{in}, (s_2)_{in}, \cdots, (s_n)_{in}\rangle
\end{aligned}
$$

The definition of C conforms to definition 2.5 and so it represent a state graph.

We are interested in state graphs which are valid specifications of asynchronous circuits, which means that they are deterministic and switch-count correct. For them the following theorem can be established.

Theorem 3.19

The product of deterministic and switch-count correct state graphs is a deterministic and switch-count correct state graph.

Proof

From statement 3.12 it comes that the product is deterministic. Let C_i and C represent a product operand and the product result, respectively. Let ω be a semi-walk in C; ω maps back to each operand C_i as a semi-walk ω_i obtained by contracting in ω all transitions not labelled with events in E_i. Assume now C is not switch count correct. Then there is a signal v and a semi-walk ω in C, such that v is switch count incorrect in ω. But then any contraction of ω not including events $v+$ and $v-$ is switch count incorrect in signal v. Thus signal v is switch count incorrect in every operand where it appears, which contradicts the switch count correctness assumption of the product operands.

3.5 Factorization

Given a transition system G can we determine a set $\{G_i | i = 1,2,\cdots,n\}$ of transition systems such that $G \cong$ **reach**$(G_1 \times G_2 \times \cdots \times G_n)$? This is the purpose of factorization. Let introduce some theory which leads to the conditions of factorization. It is based on Morin work in (concurrent)

asynchronous systems ([59]. An asynchronous system is a pair $\langle G, I\rangle$ where $G = \langle S, E, \Theta, s_{in}\rangle$ is a transition system and $I \subseteq E \times E$ is the independence relation, an irreflexive and symmetric binary relation which satisfies the following swap condition

if $\langle e_1, e_2\rangle \in I$ **then**

$$\Big(\langle s_1, e_1, s_2\rangle, \langle s_2, e_2, s_3\rangle \in \Theta \Rightarrow (\exists_{s'_2 \in S} : \langle s_1, e_2, s'_2\rangle, \langle s'_2, e_1, s_3\rangle \in \Theta)\Big)$$

An asynchronous system is called sequential if $I = \emptyset$. Morin has determined conditions which allow an asynchronous system to be decomposed into a product of sequential factors. Factors are obtained by deterministic projections — collapses in Morin terminology — of the transition system underlying the asynchronous system by a set of dependent events. Subset $\Delta \subseteq E$ is a set of dependent events if $(\Delta \times \Delta) \cap I = \emptyset$. We are not directly interested in decomposing concurrency. However Morin results can be used to determine if a set of deterministic projections are a factorization of a given transition system: a transition system G can be seen as the asynchronous system $\langle G, \emptyset\rangle$ and so any Δ is a set of dependent events. We are reproducing here his theorem which determines those conditions. Some adaptations were done in order to accomodate it to our purpose and to our terminology.

Let $G = \langle S, E, \Theta, s_{in}\rangle$ be a transition system; let $E_1, \cdots, E_n \subset E$; let $G_i = G \Downarrow E_i$, for $i = 1, \cdots, n$; let $\sim_i$ be the equivalence relation associated to projection G_i; let $\mathcal{G} = \{G_i | i = 1, \cdots, n\}$ be a set of projections:

Definition 3.20 (state-state-separation)

Set $\mathcal{G}$ holds the state-state separation property if and only if

$$\forall s_1, s_2 \in S, s_1 \neq s_2 \Rightarrow (\exists_{G_i} : [s_1]_{\sim_i} \neq [s_2]_{\sim_i})$$

Definition 3.21 (state-event-separation)

Set $\mathcal{G}$ holds the state-event separation property if and only if

$$\forall s \in S, \forall e \in E, s \overset{\mathcal{G}}{\not\rightarrow} \Rightarrow (\exists_{G_i} : [s]_{\sim_i} \overset{\mathcal{G}}{\not\rightarrow})$$

where $u \overset{\mathcal{G}}{\not\rightarrow}$ means event e is not enabled in state u.

Theorem 3.22 (Morin's decomposition)

If the set $\mathcal{G}$ holds both separation properties then

$$\mathbf{reach}(G) \cong \mathbf{reach}(G_1 \times \cdots \times G_n)$$

Let G be a transition system and let $\mathcal{G} = \{G_i \mid G_i = G \Downarrow E_i, \text{for } i = 1, 2, \cdots, n\}$ be a set of projections, such that $E_1 \cup E_2 \cup \cdots \cup E_n = E$. In theorem 3.18 we have shown that there is an homomorphism from **reach**(G) into **reach**$(G_1 \times G_2 \times \cdots \times G_n)$. The separation conditions described above are closely related to the injectivity and surjectivity of this homomorphism.

Theorem 3.23 (injectivity)

$\mathcal{G}$ holds the state-state separation property if and only if there is an injective homomorphism from G into **reach**$(G_1 \times \cdots \times G_n)$.

Proof

Let $|\mathcal{G}| = 2$; for sets of projections with more elements a similar reasoning can be done. Let the pair $\langle \sigma, \tau \rangle$ represent the homomorphism, where σ is the function on states and τ the function on transitions. Then $\sigma(s) = \langle [s]_{\sim_1}, [s]_{\sim_2} \rangle$ where $\sim_1$ and $\sim_2$ are the equivalent relations associated to the two projections.

Assume now state-state separation does not hold. Thus there are 2 states $s, s' \in S$ such that $s \neq s'$, $[s]_{\sim_1} = [s']_{\sim_1}$ and $[s]_{\sim_2} = [s']_{\sim_2}$. Then $\sigma(s) = \langle [s]_{\sim_1}, [s]_{\sim_2} \rangle = \langle [s']_{\sim_1}, [s']_{\sim_2} \rangle = \sigma(s')$ and the homomorphism is not injective, which proves sufficiency.

Now consider the homomorphism is not injective. Then $\sigma(s) = \sigma(s')$ for some $s \neq s'$. Thus we have $\langle [s]_{\sim_1}, [s]_{\sim_2} \rangle = \langle [s']_{\sim_1}, [s']_{\sim_2} \rangle$ and state-state separation property does not hold. Necessity is also proved.

Theorem 3.24 (surjectivity)

$\mathcal{G}$ holds the state-event separation property if and only if there is a surjective homomorphism from G into $G_\times =$ **reach**$(G_1 \times \cdots \times G_n)$.

Proof

Let the pair $\langle \sigma, \tau \rangle$ represent the homomorphism, where σ is the function on states and τ the function on transitions; let $G = \langle S, E, \Theta, s_{in} \rangle$;

Assume the homomorphism is not surjective. There two cases to consider: σ not surjective and τ not surjective. Since the product is taken up to reachability, if σ is not surjective, τ also is not. So it is enough to consider τ not surjective. Then there are a state $s \in S$ and an event $e \in E$ such that $s \stackrel{e}{\not\rightarrow}$ but $\sigma(s) \stackrel{e}{\rightarrow}$. If $\sigma(s) \stackrel{e}{\rightarrow}$ then, for every i, either $[s]_{\sim_i} \stackrel{e}{\rightarrow}$ or $e \notin E_i$. Thus state-event separation does not hold.

Assume now state-event separation does not hold. There is a state s and an event e such that $s \stackrel{e}{\not\rightarrow}$ and for every projection G_i — with equivalent relation $\sim_i$ — which not hide event e, $[s]_{\sim_i} \stackrel{e}{\rightarrow}$. But then $\langle \sigma(s), e, y \rangle$, for some state y, is a transition in $G_\times$. Thus the homomorphism is not surjective, which concludes the proof.

3.6 Event Insertion on Transition Systems

A basic transformation of a transition system is accomplished by the insertion of a new event. There can be different schemes for the insertion to take place. However, and in general, we are interested in a scheme which preserves some of the properties of the original transition system, like trace equivalence, persistence, and others.

A common scheme is based on a subset of states, whose exiting events are delayed by the new event. Let G be a transition system, S its set of states and $r \subset S$. The insertion takes place as follows:

- $G[r]$, the transition subsystem induced by r, is doubled;
- transitions entering $G[r]$ are kept unchanged;
- transitions exiting $G[r]$ have their source states transferred to corresponding states in the replica;
- all states in $G[r]$ are connected via the new event to the corresponding states in the replica.

Thus the passage from r to $S - r$ is delayed by the new event. This insertion is illustrated by figure 3.10.

Definition 3.25 (event-insertion)
Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system; let $x \notin E$ be a new event; let $r \subseteq S$ be an arbitrary subset of states; let r', $r' \cap S = \emptyset$, be a set of new states such that for each $s \in r$ there is one and only one $s' \in r'$ and vice versa, that is, we can define bijection $\imath : r \rightarrow r'$. The insertion of x in G by

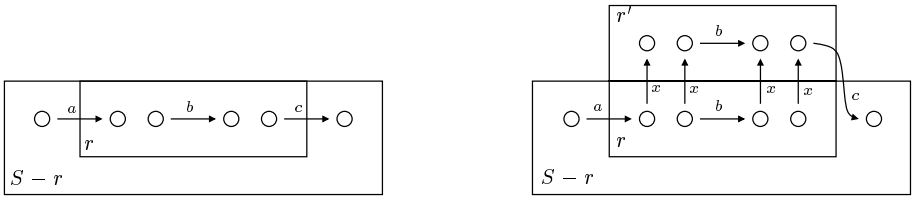


Figure 3.10: Illustrating insertion of a new event on a transition system: the transition subsystem induced by r is doubled; transitions going from a state in r to a state in $S - r$ are transformed into transitions going from a state in r' into a state in $S - r$; states in r are connected to corresponding states in r' via a transition labeled with the new event x.

r produces another transition system $G' = \langle S', E', \Theta', s_{in} \rangle$ defined as follows:

$$
\begin{aligned}
S' &= S \cup r' \\
E' &= E \cup \{x\} \\
\Theta' &= \Theta - \\
&\quad \{\langle s_u, e, s_v \rangle \mid s_u \in r \ \wedge \ s_v \in S - r\} \cup \\
&\quad \{\langle \imath(s_u), e, \imath(s_v) \rangle \mid s_u, s_v \in r \ \wedge \ \langle s_u, e, s_v \rangle \in \Theta\} \cup \\
&\quad \{\langle \imath(s_u), e, s_v \rangle \mid s_u \in r \ \wedge \ s_v \notin r \ \wedge \ \langle s_u, e, s_v \rangle \in \Theta\} \cup \\
&\quad \{\langle s_u, x, \imath(s_u) \rangle \mid s_u \in r\}
\end{aligned}
$$

Trace equivalence, determinism and deadlock-freedom are preserved by construction. The same does not happen with persistence and commutativity. In [23] two theorems were proposed and proved which determine conditions that the set r must satisfy in order persistence and commutativity to be preserved.

Definition 3.26 (SIP-set)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system, $x \notin E$ be a new event, and $r \subseteq S$. Let $G' = \langle S', E', \Theta', s_{in} \rangle$ be the transition system obtained by the insertion of x in G by r.

1. *r is said to be a* persistence preserving set *if for all $e \in E$, if e is persistent in G then e is persistent in G';*

2. *r is said to be a* speed-independence preserving *set (often shortened SIP-set) if, additionally, if G is commutative then G' also is.*

Theorem 3.27 (persistence preserving)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system and $r \subseteq S$: r is a persistence preserving set if and only if

$$(\langle s_a, e_1, s_b \rangle, \langle s_a, e_2, s_c \rangle, \langle s_c, e_1, s_d \rangle \in \Theta \ \wedge \ s_c \in r, s_d \notin r) \Rightarrow (s_a \in r \ \wedge s_b \notin r)$$

Theorem 3.28 (speed-independence preserving)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a commutative transition system and $r \subseteq S$: r is a SIP-set if and only if r is a persistence preserving set and

$$(\langle s_a, e_1, s_b \rangle, \langle s_a, e_2, s_c \rangle, \langle s_c, e_1, s_d \rangle, \langle s_b, e_2, s_d \rangle \in \Theta \ \wedge \ s_a, s_b \in r \ \wedge \ s_c \notin r) \Rightarrow s_d \notin r$$

3.7 Signal Insertion on State Graphs

Quite often the transformation to be executed in a state graph corresponds to the insertion of one or more signals. The insertion of each signal actually corresponds to the insertion of two new events, the positive and negative transitions of the signal to be inserted. After signal insertion we should obtain a valid state graph, which preserves the properties of the original state graph. This means that, in addition to property preserving considerations of the underlying transition system, consistency must be preserved, both in terms of the original signals and of the inserted ones. Event insertions based on definition 3.25 preserve consistency of the original signals. We have to obtain conditions which guarantee consistency of the new added signal.

Let insert the new signal v in a state graph G based on two subsets of states, S^+ and S^-, such that $S^+ \cap S^- = \emptyset$. After insertion of $v+$ by S^+ and of $v-$ by S^-, G is transformed to state graph G', depicted in figure 3.11.a in a compact form. In order signal v to be consistent, the set $S - S^+ - S^-$ should be composed of two disconnected sets, one of them being only reachable from $S^{-\prime}$ and the other only reachable from $S^{+\prime}$. This is illustrated in figure 3.11.b, where it is clear that is impossible to pass twice through $v+(v-)$ without passing through $v-(v+)$. Thus, signal insertion is actually based on a 4 blocks partition, $I = \langle S^0, S^+, S^1, S^- \rangle$. Partition I is usually referred to as an I-partition [77]. Next theorem determines conditions the I-partition must hold in order the signal insertion results in a consistent state graph.

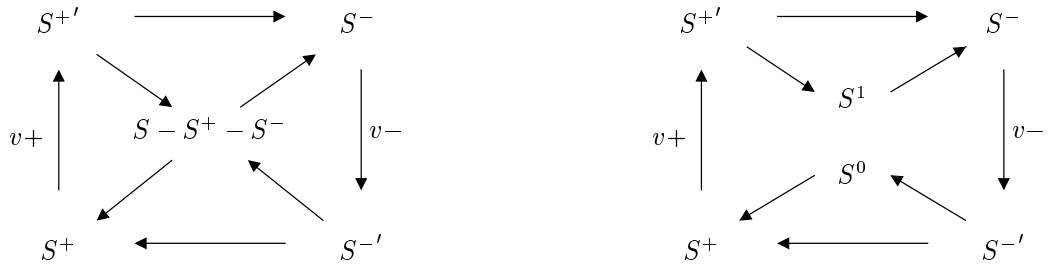


Figure 3.11: Transformed state graph after insertion of a new signal v based on the set of events S^+ and S^-.

Theorem 3.29 (consistent signal insertion)

Let G be a consistent state graph and $I = \langle S^0, S^+, S^1, S^- \rangle$ be an I-partition of G. The state graph obtained after insertion of a new signal v is consistent iff the only allowed transitions between partition blocks are the following: $S^0 \rightarrow S^+ \rightarrow S^1 \rightarrow S^- \rightarrow S^0$, $S^+ \rightarrow S^-$ and $S^- \rightarrow S^+$.

3.8 Conclusions

Transition systems and state graphs are used to represent the behavior of asynchronous circuits. These specifications will be analysed, in order to find out desired patterns, and will be manipulated in order to make them earn some desired properties. In this chapter we introduced the main operations on transition systems and state graphs needed to carry out both analysis and manipulation.

We started with morphism, an operation which maps a transition system into another transition system. A morphism can be defined to hide one or more events of a transition system or to prove the equivalence between transition systems. Then, we introduced the asynchronous product of two or more transition systems. This operation will assume a preponderant role in chapter 5, when we develop our synthesis procedure. We concluded the chapter, reviewing a fundamental operation on state graphs, the insertion of a new signal, which, in turn, corresponds to a fundamental operation in transition systems, the insertion of a new event. Here we have just borrowed definitions from the literature. In chapter 5 we will adapt these definitions to our reality.

Chapter 4

Conflicts

In chapter 1 conditions for the implementability of a specification as a speed-independent asynchronous circuit were presented. Apart from switch-count correctness, a specification must hold the complete state coding (CSC) property and be persistent with respect to all signals, except for inputs being disabled by other inputs. In this last case, the responsibility for the disabling is of the environment, which is assumed to be well behaved.

Configurations which invalidate the implementability as a speed-independent circuit are called *conflicts*. Two types of conflicts can be referred: *complete state coding (CSC) conflicts* and *speed independent (SI) conflicts*. A CSC conflict exists whenever two or more states violate the CSC property, that is, there are different states with the same binary code but with different sets of excited output signals. Figure 4.1.a shows a state graph with two CSC conflicts. Assume a is the input and b the output signals. States x_1 and x_5 have the same binary code, 10. However they have different sets of excited output signals. Signal transition $b+$ is enabled in x_1 but not in x_5. Similarly, states x_2 and x_4 have the same binary code, 01, with signal transition $b-$ enabled in x_4 but not in x_2.

Many approaches to solve CSC conflicts have been proposed (e.g. [17, 21, 82, 78]). They can solve most conflict situations, but, in some cases, restrict the class of acceptable specifications. The one described in [17], and implemented in the software synthesis tool *petrify* [22, 20], can automatically solve most CSC conflicts while accepting quite generic specifications, even with a large amount of states. Based on the theory of regions, it works doing behavior-preserving transformations on the state graph specification. The CSC conflicts are solved by means of insertions of new internal signals,

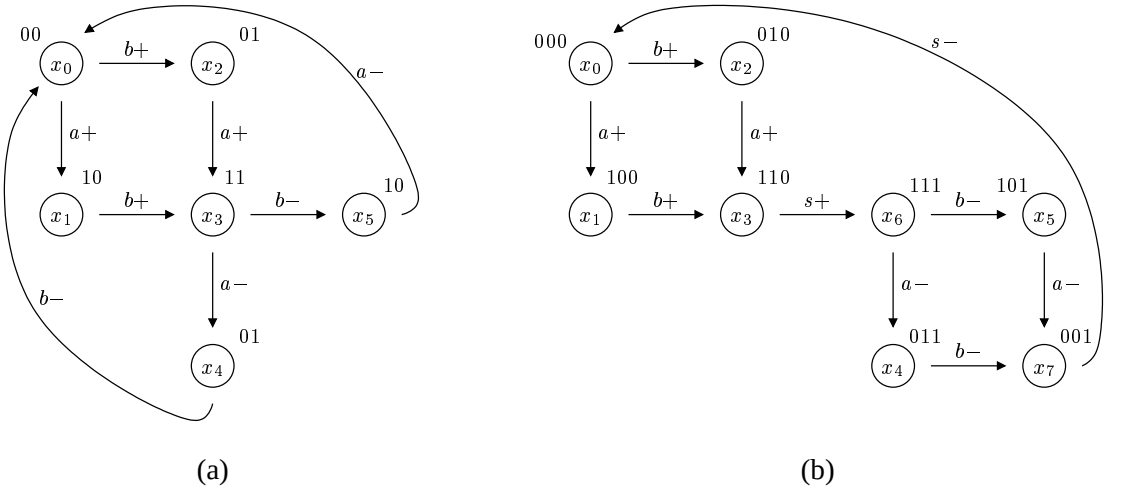


Figure 4.1: (a) A state graph with two CSC conflicts, between states x_1 and x_5 and between states x_2 and x_4. (b) Conflict free solution obtained by *petrify*.

which must assume different values for the states under conflict. Using the synthesis tool *petrify* to solve the CSC conflicts of the state graph in figure 4.1.a we obtain the state graph in figure 4.1.b. Both conflicts are solved by adding the single internal signal s.

Not all CSC conflicts are solved by *petrify*. Consider for instance the state graph in figure 4.2, borrowed from [81]. There is a CSC conflict between states x_7 and x_{13}. When committed to work this specification, *petrify* reports its unability to find a solution, because the "input non-commutativity produces irreducible CSC conflicts". In state x_1 both events a_1+ and a_2+ are enabled. Depending on the order of their occurrence either state x_7 or x_{13} is reached. Actually, this non-commutativity is the cause of the CSC conflict.

Non-commutativity is one kind of SI conflict, as we will present in the next section. In the previous example, while the SI conflict persists, the CSC conflict cannot be solved; eventually, solving the SI conflict will make the CSC conflict to disappear.

Thus, elimination of SI conflicts must be addressed before treating CSC conflicts. The rest of this chapter is dedicated to SI conflicts. We will start by introducing the notion of *concurrent conflict*, a conflict associated with events simultaneously enabled in a given state. Then, concurrent conflicts will be analyzed in the field of state graphs, considering their association with input and output sig-

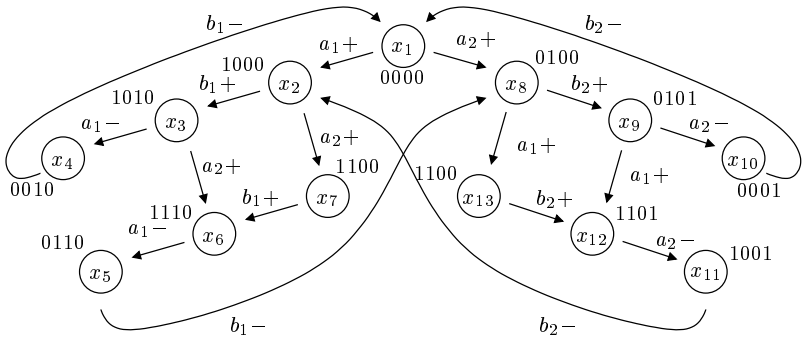


Figure 4.2: A state graph with an irreducible CSC conflict.

nals. Some configurations, as it will be shown, invalidate the implementation as a speed-independent circuit. In such cases, the concurrent conflict is called a *speed-independent (SI) conflict*.

4.1 Concurrent Conflicts

Concurrent conflicts are associated with situations where the relative speed of simultaneously enabled events dictates how the system evolves. Two classes of situations can be considered: event non-persistence and event non-commutativity. An *event non-persistence* exists whenever an enabled event is disabled by the occurrence of another event. For instance, in figure 4.2, in state x_3, both events a_1- and a_2+ are enabled. The occurrence of a_1- leads the system to state x_4, where a_2+ is not enabled anymore. An *event non-commutativity* exists whenever the order of firing of two simultaneously enabled events, leads the system to different states. As already referred before, in figure 4.2, there is a non-commutativity situation between events a_1+ and a_2+ in state x_1.

Before giving formal definitions of concurrent conflicts let us make a review of previous incursions on the same subject. In [79] a marking M of a Petri net is called a *conflict marking* if there exist two events, a and b, enabled in M and one of them is not enabled in the marking reached after the firing of the other, that is, b is not enabled in $M_a, M \xrightarrow{a} M_a$. Consider for instance the Petri net in figure 4.3. In the initial marking, $M_0 = \{p_2\}$, both transitions $b*$ and $c*$ are enabled. The firing of $b*$ makes the system evolve to marking $M_1 = \{p_1\}$, where $c*$ is not enabled. On the other hand, the firing of $c*$ makes the system evolve to marking $M_2 = \{p_3\}$, where $b*$ is not enabled. Thus, M_0 is a conflict

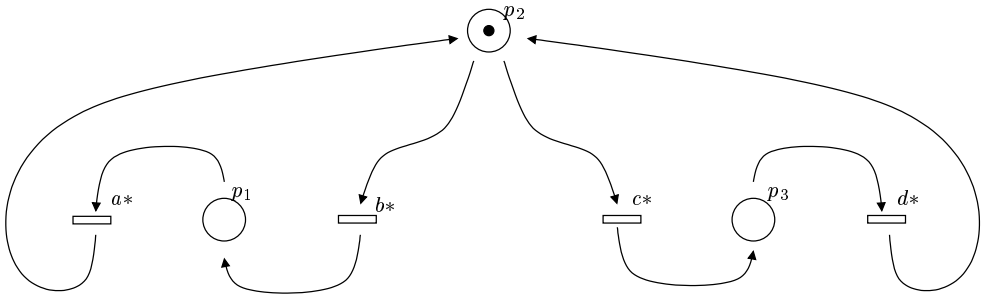


Figure 4.3: A Petri net with a conflict marking.

marking. The definition of Petri net used in [79] corresponds to our definition of single event Petri net. There is a one-to-one correspondence between transitions and events. If a transition becomes disabled the same happens to the labeling event.

Also in [79] a state x of a state graph is called a *conflict state* in signal v, if v is excited in x and there exists a state $x\prime$, immediately reachable from x, where v has the same value and is not excited, i.e., there is a disabling of signal v when moving from x to $x\prime$. State x_3 of the state graph depicted in figure 4.2 is a conflict state in signal a_2. Signal a_2 is excited in that state but after the firing of a_1- state x_4 is reached and a_2 is not excited there.

In [25] no formal definition of conflict is given. However conflicts in Petri nets are associated with places, instead of markings. A place p of an STG (Petri net) is called a *conflict place* if

1. there are n tokens in p; and
2. there are m transitions simultaneously enabled by p and $m > n$.

Thus, $m - n$ transitions become disabled after the firing of the other n. Place p_2 of the Petri net in figure 4.3 is a conflict place. In the initial marking both transitions $b*$ and $c*$ are enabled and p_2 is a pre-condition for both. However, the single token in p_2 is not enough to fire both transitions.

Note that the indication "simultaneously enabled" is crucial in the definition above. Consider for instance the Petri net in figure 4.4.a. Place p_2 is in the pre-set of both transitions e_1 and e_2. But no marking enables simultaneously both events. Thus, p_2 is not a conflict place according to the

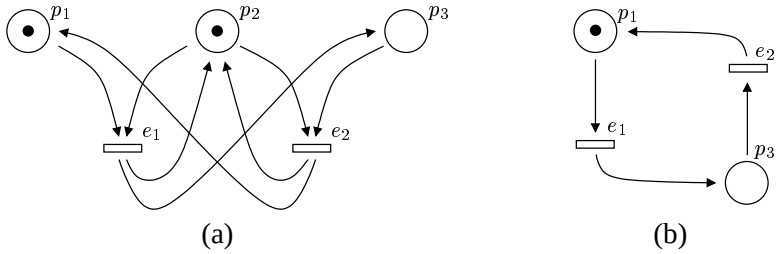


Figure 4.4: Two behavior equivalent Petri nets. The one on the left has a false conflict place.

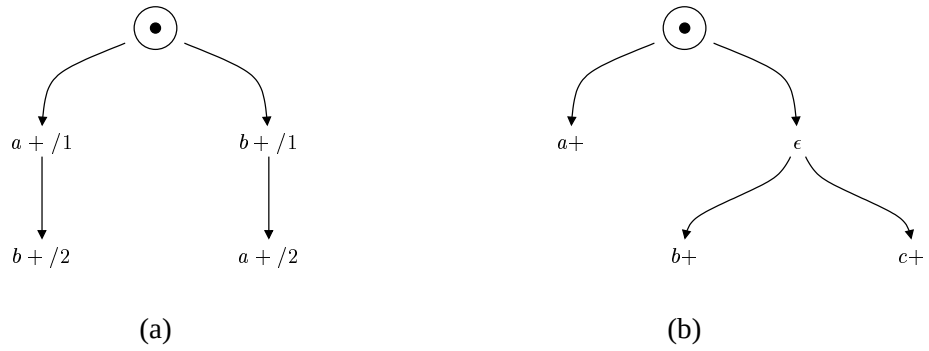
definition. Actually, place p_2 is redundant, and the Petri net in figure 4.4.a is equivalent to the one in figure 4.4.b.

In the previous definitions conflicts are associated with markings, states or places and not with transitions or events. This is done in [49], where we can find a deep analysis on conflicts, based on Petri net unfolding. Two transitions are in a *direct conflict* if there exists a reachable marking M, where both transitions are enabled and the firing of one of them leads to a marking where the other is disabled. Transitions $b*$ and $c*$ of the Petri net in figure 4.3 are in direct conflict, while transitions e_1 and e_2 of the Petri net in figure 4.4 are not.

This definition is based on a disabling function between transitions, while we are more interested in events (signal transitions) and thus on a disabling function between events. Consider for instance the STG fragment depicted in figure 4.5.a. In it indices are used to distinguish different instances of the same event. Thus $a+/1$ and $a+/2$ are different instances of the event $a+$. Transitions $a+/1$ and $b+/1$ are in direct conflict, but events $a+$ and $b+$ are not. Because of that, in [49], a direct conflict between two transitions is called *fake* if there is not a disabling function between the signal transitions labeling the transitions.

The authors go deeper in the conflict analysis, showing that there can exist a disabling function between signal transitions where no direct conflict exists involving them. That's because of the possible existence of *dummies* events labeling the transitions. In the fragment of an STG depicted in figure 4.5.b, there is no direct conflict between $a+$ and $b+$, but these events disable each other.

This problem of non-locality in conflict detection makes it difficult to find out conflicts on STG descriptions. In SG descriptions conflicts are a local problem because dummies do not appear there.



(a) (b)

Figure 4.5: (a) A direct conflict which does not produces an event disabling. (b) A direct conflict involving a dummy event, which produces a disabling between events.

For that reason we have decided to address conflicts and conflict specifications at SG level, leaving the STG level for a future opportunity.

There is a diversity of definitions for conflict, all related to the same phenomenon. We will introduce our set of definitions that tries to merge all the previous diversity. The definitions will be given at state level.

Definition 4.1 (non-persistence)

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a transition system. Two events $a, b \in E$ are said to be in a non-persistence relation *in state $s \in S$ if and only if*

$$\langle s, a, s_a \rangle, \langle s, b, s_b \rangle \in \Theta \ \wedge \ (\langle s_a, b, s_{ab} \rangle \notin \Theta \ \vee \ \langle s_b, a, s_{ba} \rangle \notin \Theta)$$

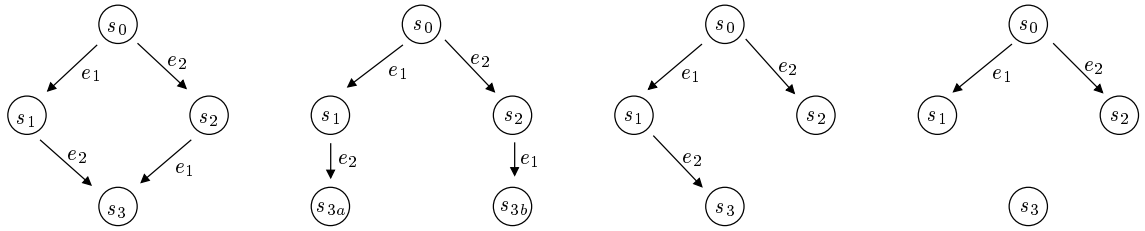
If $\langle s_a, b, s_{ab} \rangle \notin \Theta \ \wedge \ \langle s_b, a, s_{ba} \rangle \notin \Theta$ the non-persistence is called symmetric*; otherwise it is called* asymmetric.

If none of the disabling occurs a and b are persistent in state s. Which means states s_{ab} and s_{ba} exist. But they can be different states. When such a case occurs a and b are said to be in a *non-commutative* relation in state s. Formally:

Definition 4.2 (non-commutativity)

Two events $a, b \in E$ are said to be in a non-commutative relation *in state $s \in S$ if and only if*

$$\langle s, a, s_a \rangle, \langle s, b, s_b \rangle, \langle s_a, b, s_{ab} \rangle, \langle s_b, a, s_{ba} \rangle \in \Theta \ \wedge \ s_{ab} \neq s_{ba}$$



(a) (b) (c) (d)

Figure 4.6: Signal transitions e_1 and e_2 are: (a) purely concurrent (persistent and commutative); (b) persistent but non-commutative; (c) asymmetrically non-persistent; (d) symmetrically non-persistent.

Conflict relations between signal transitions can now be defined in terms of non-persistence and non-commutativity relations.

Definition 4.3 (concurrent conflict)

Two events $a, b \in E$ are said to be in a concurrent conflict relation *at state $s \in S$ if and only if one of the following conditions hold:*

1. *a and b are in a non-persistence relation at s.*
2. *a and b are in a non-commutative relation at s.*

Events a and b are in concurrent conflict *in transition system G if and only if there is a state $s \in S$ where they are in a conflict relation.*

When no conflict relation exist we say events a and b are *purely concurrent*, which correspond to a commutative persistence relation. Figure 4.6 shows the four types of concurrent relations that can exist between two events. In case (d) state s_3 can eventually not exist.

Our interest in conflicts has a primary concern: the implementation of a state graph specification as a speed-independent circuit. In that sense, concurrent conflicts in state graphs must be detected and conditions that invalidate the implementation as a speed-independent circuit must be identified. A concurrent conflict in a state graph is called a *speed-independent* (SI) conflict if it invalidates the implementation as a speed-independent circuit.

The description of a circuit includes its behavior, as well as the behavior of the environment where the circuit is immersed. Correctness of circuit behavior must be ensured from the point of view of its interaction with the environment. As long as the environment behaves correctly the circuit should react correctly. But there is a question that should be put: can the environment behave correctly? On one hand, it can be assumed that is up to the designer, who has specified the circuit description, to ensure correct behavior of the environment. In that sense, during circuit implementation it is assumed the environment behaves correctly, no matter how that correctness is accomplished. On the other hand, one can wonder about the possibility of any anomaly in the environment behavior. Eventually, the environment is also a circuit and so considerations put in circuit synthesis should also be put to the environment.

An analysis of concurrent conflicts in state graphs follows. In that analysis we examine the different types of conflicts – symmetric and asymmetric non persistencies and non-commutativities – and the different types of signals – input, output, and internal signals.

4.2 Non-Persistencies

4.2.1 Symmetric Non-Persistence

Let $\langle S, E, \Theta, s_{in} \rangle$ be a state graph, where $E = V \cup \{+, -\}$, being V the set of signals. Let $a*, b* \in E$ be two signal transitions, associated with signals a and b, involved in a symmetric non-persistency relation in some state $s \in S$. Figure 4.7 shows a state graph with such a conflict, where x_1, $y+$, and $z+$ play the roles of respectively s, $a*$, and $b*$.

If a and b are both input signals, it is up to the environment where the circuit is immersed, to decide whether to fire $a*$ or $b*$; as long as the environment behaves correctly the circuit reacts correctly. From the point of view of the environment $a*$ and $b*$ are outputs. It is reasonable to assume that the environment can choose which output to fire. Thus, symmetric non-persistencies among input signals are not SI conflicts.

If we assume y and z are input signals and x is an output signal, figure 4.7 represents a specification with a symmetric non-persistence between two input signal transitions. In that case, figure 4.8.a shows a possible circuit implementation. Looking only at this circuit, we can identify a situation with

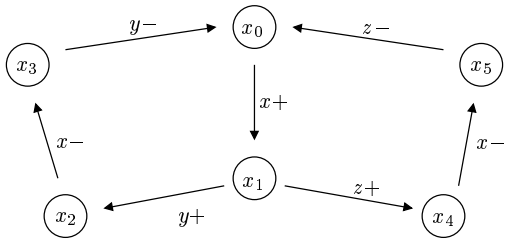
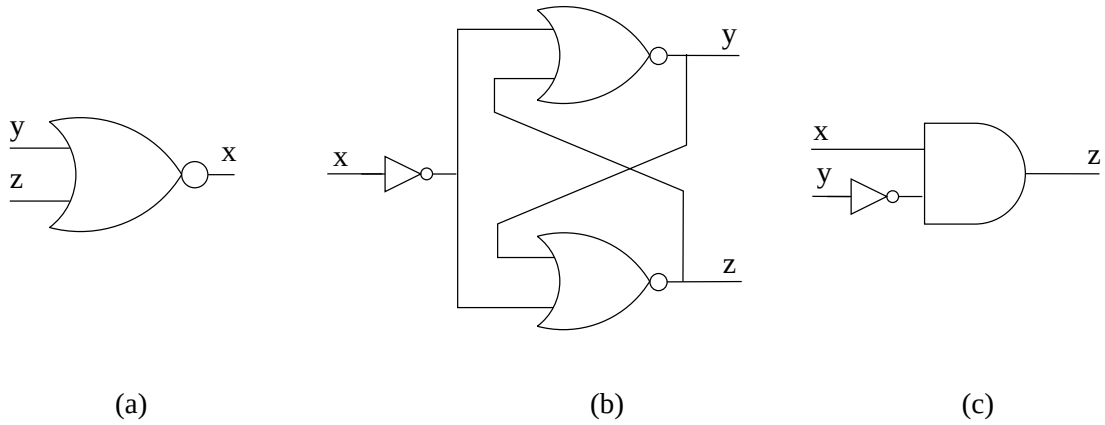


Figure 4.7: A state graph specification with a symmetric non-persistency relation between signal transitions $y+$ and $z+$ in state x_1.



(a) (b) (c)

Figure 4.8: Implementations for the specification depicted in figure 4.7, assuming different types for the x, y and z signals: (a) circuit implementation assuming y and z as inputs and x as output; (b) non-speed independent circuit implementation assuming x as input and y and z as outputs; (c) non-speed independent circuit implementation assuming x and y as inputs and z as output.

a possible malfunction: if y is at one, z at zero and x at zero and the environment decides to change, almost simultaneously, y to zero and z to one, it could appear a spike at the output. But, according to the specification, the environment never behaves like that. When in state x_3, the one corresponding to the situation described above, the environment fires $y-$, waits until the circuit rises the output and only then it can change z to one.

Consider now that a and b are output signals. When the circuit is in state s is up to it to decide which event to fire, $a*$ or $b*$. Since both events are in equal conditions to fire, and since in real circuits signal transitions are not instantaneous, both $a*$ and $b*$ could start to fire at the same time. But only one can

actually do it and so the other must move backwards. This “fight” between the two events is known to result in some anomalous behavior, like meta-stability or oscillation.

Figure 4.8.b, shows a circuit suffering from this anomalous behavior: it is a non-speed independent implementation of the specification in figure 4.7, assuming x as an input signal and y and z as output signals. Consider the system at state x_0, with the input and the outputs at 0. The outer inputs of the NOR gates are at 1, while the inner inputs are at 0. If x changes from 0 to 1, the outer inputs of the NOR gates change to 0. Then the outputs of the NOR gates are changing to 1, but at the same time the inner inputs are changing to 1. For a CMOS implementation this may result in one of two anomalous behaviors for the outputs: an oscillation between the two logical values, 0 and 1; an unstable equilibrium in an intermediate value, the meta-stable state. Eventually, after an undetermined time the system evolves to a stable state, with one of the outputs at 1 and the other at 0, i.e., to either state x_2 or x_4. The same conclusions are attained if instead of outputs we consider internal signals.

Thus, there is no pure digital implementation for such a behavior. But, is there a solution holding the speed-independent assumption, eventually using non-digital devices? The mutual exclusion element (mutex), presented in chapter 1 and depicted in figure 1.22, has the state graph description shown in figure 4.9. If we short circuit the two inputs, we get a circuit with one input r and two outputs g_1 and g_2, whose state graph description is given by figure 4.9.b. This is exactly the behavior we want to implement. So, we get a speed-independent implementation for our behavior.

A symmetric non-persistency between non-input signal transitions is a SI-conflict. But, eventually, using special devices, like the mutex, we can get a speed-independent implementation for a behavior with a symmetric non-persistency between non-input signal transitions.

Finally consider that one of the signals, say a, is an input signal and the other, b, is an output signal. According to the behavior, if the circuit decides to fire the output transition, the environment should not fire the input transition. Also according to the behavior, if the environment decides to fire the input, the circuit should not fire the output. But the environment can only sense that the circuit is firing the output when it starts doing it. Similarly, the circuit can only sense that the environment is firing the input when it starts doing it. So, there is the real chance that both events start occurring at the same time. Who should move backwards in such a case, the circuit or the environment? Maybe both.

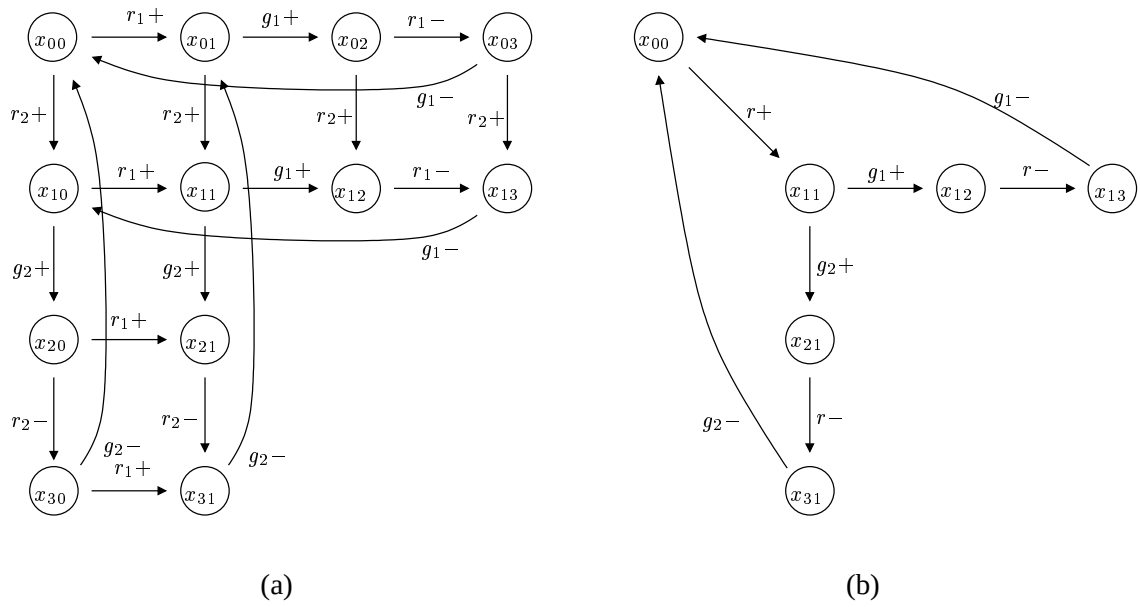
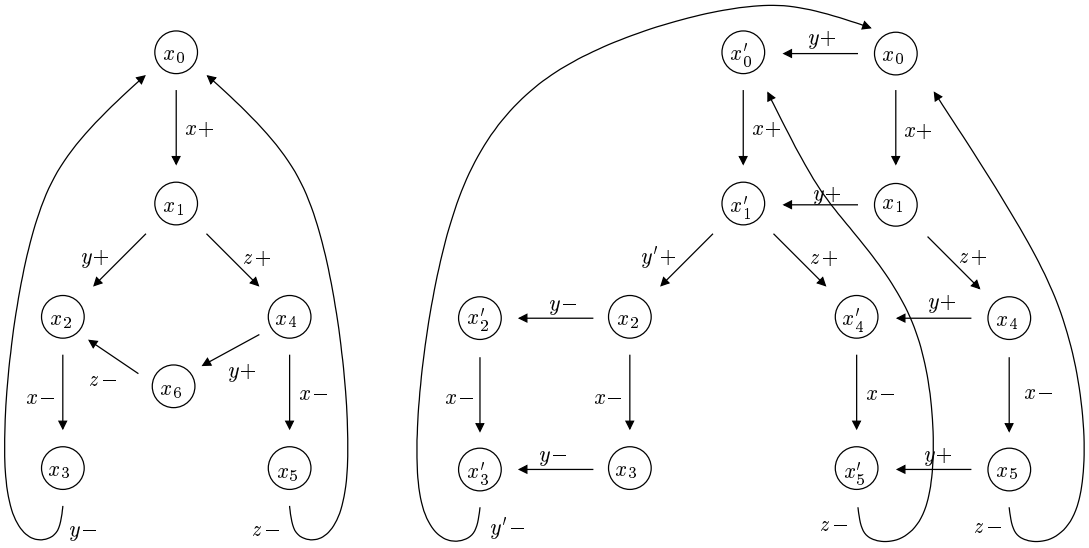


Figure 4.9: (a) State graph for the mutual exclusion element (mutex). (b) State graph for a mutex with the inputs short circuited.

Consider, as an example, the specification in figure 4.7, assuming x and y as inputs and z as an output. A possible circuit implementation, assuming the environment behaves correctly, doesn't matter how, is depicted in figure 4.8.c. If the circuit is at state x_0 and x changes to 1, the unstable state x_1 is reached, where the inputs of the AND gate are at 1 and the output at 0. Thus, the output starts changing to 1. If by the same time the environment decides to fire $y+$, the lower input of the AND gate changes to 0 and the changing in the output is stopped. If, eventually, the z output reaches the logical value 1, the behavior is not observed.

There are different possible decisions for such a behavioral description. We can refuse the description, suggesting to the designer to make some modifications to it. For instance, in the description of figure 4.7, a new state, backward connected to x_4 by $y+$ and forward connected to x_2 by $z-$, can be added, transforming the symmetric non-persistency between an input and an output into an asymmetric non-persistency, where an input disables an output, and a symmetric non-persistency between two inputs (see figure 4.10.a). Asymmetric non-persistencies are covered below. Alternatively, we can try to fix the problem from the circuit point of view. For instance, we can consider that the signal



(a) (b)

Figure 4.10: Two different alternatives to overcome the symmetric non-persistency in figure 4.7, when assuming x and y as inputs and z as output: (a) the original non-persistency is transformed into an asymmetric non-persistency, where an input signal transition disables an output signal transition, and a symmetric non-persistency between input signal transitions; (b) by stretching the input signal, the original non-persistency is transformed into a symmetric non-persistency between non-input signal transitions.

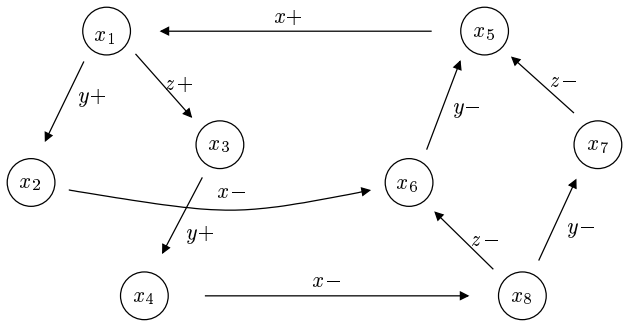


Figure 4.11: A state graph specification with an asymmetric non-persistency relation between signal transitions $y+$ and $z+$ in state x_1, such that the occurrence of $y+$ disables $z+$.

issued by the environment is not the same perceived by the circuit, and decompose the input y into two different signals. This would result in the state graph depicted in figure 4.10.b, where y is the input and y' an internal signal. In this way the conflict is converted into a symmetric non-persistency between an output and an internal signal.

4.2.2 Asymmetric Non-Persistence

Let $\langle S, E, \Theta, s_{in} \rangle$ be a state graph, where $E = V \cup \{+, -\}$, V being the set of signals. Let $a*, b* \in E$ be two signal transitions, associated with signals a and b, involved in an asymmetric non-persistency relation in some state $s \in S$, such that b is disabled by a. Figure 4.11 shows a state graph with such a conflict, where x_1, $y+$, and $z+$ play the roles of respectively s, $a*$, and $b*$. Similarly to what we have done for symmetric non-persistencies, we will analyze the specification for different combinations of the signals under conflict. Four cases are to be considered: a and b as input signals, a as input and b as non-input, a as non-input and b as input, and a and b as non-inputs.

If a and b are both input signals, it is up to the environment to guarantee the correct occurrence of the conflict events. If the environment chooses to fire $b*$, it can also fire $a*$; otherwise it should fire only $a*$. It is acceptable to assume that the environment can behave that way. Figure 4.12.a shows a speed-independent circuit implementation, which follows the behavior in figure 4.11, assuming y and z as input signals and x as an output signal.

Consider now that, in the state graph of figure 4.11, a and b are output signals. Both events, $a*$ and

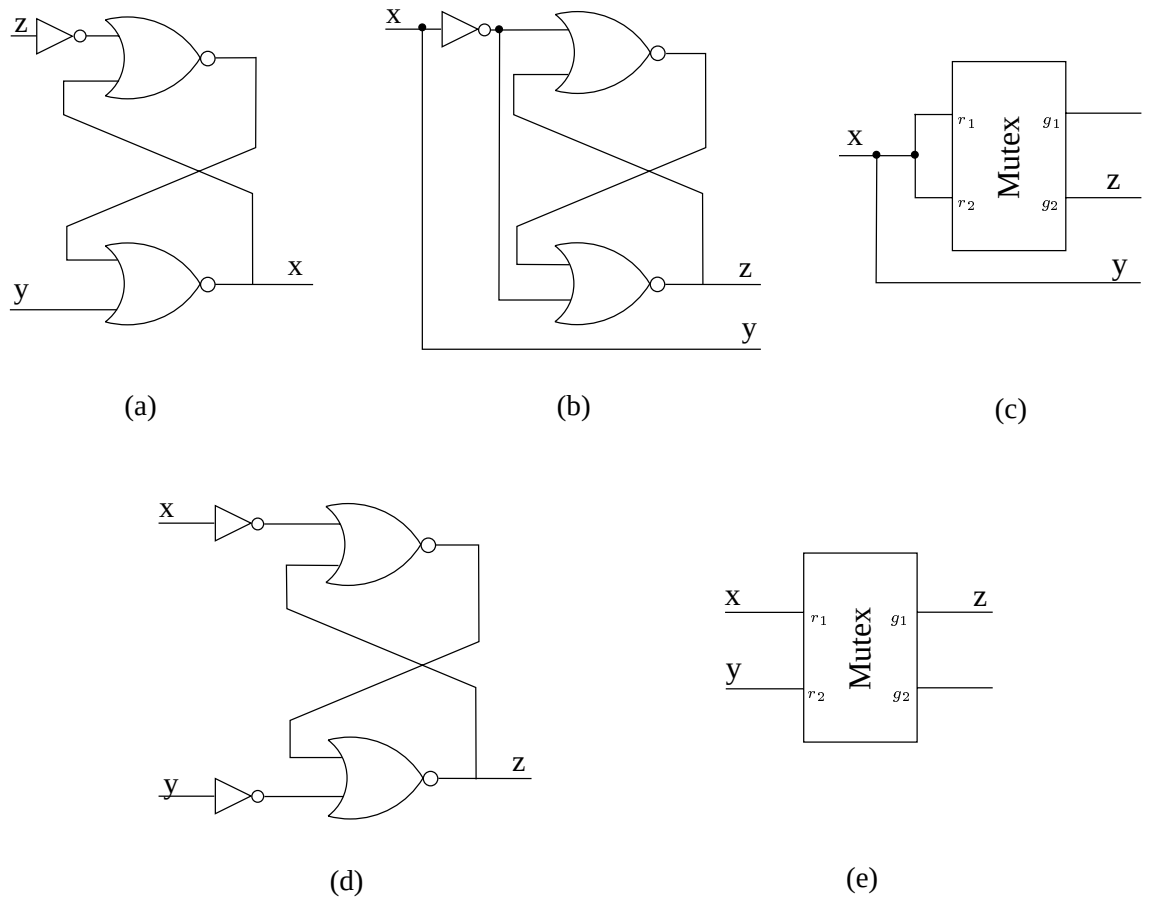


Figure 4.12: Implementations for the specification depicted in figure 4.11, assuming different types for the x, y and z signals: (a) speed-independent circuit implementation assuming y and z as inputs and x as output; (b) non-speed independent circuit implementation assuming x as input and y and z as outputs; (c) speed-independent implementation under the same assumption; (d) non-speed independent circuit implementation assuming x and y as inputs and z as output; (e) speed-independent implementation under the same assumption.

$b*$, are in conditions to fire and one of them, $a*$, will always do. However, in real circuits, because events are not instantaneous, event $b*$ can start to fire and be forced to move backwards. This can result in some anomalous behavior associated with signal b.

Figure 4.12.b shows a non-speed independent circuit implementation, which follows the behavior in figure 4.11, assuming y and z as output signals and x as an input signal. Consider that the circuit is at the stable state x_5 with the input and the outputs at 0. If the input changes to 1, the outer inputs of the NOR gates change to 0. Then the outputs of the NOR gates enter an undefined situation until one of them stabilises at 1 and the other stabilises at 0. As already mentioned before, for a CMOS implementation, this undefined situation means oscillation or meta-stability.

There is no pure digital implementation for such a behavior. But there is a possible implementation using the mutual exclusion element. Note that the circuit in figure 4.12.b, from x to z, is exactly the same as in figure 4.8.b. Thus, output signal z can be implemented as a speed-independent circuit using a mutex with the two inputs short circuited (see figure 4.12.c).

The next case corresponds to consider a as an output signal and b as an input signal. According to this behavior, if the circuit decides to fire $a*$, the environment should not fire $b*$. For that, the environment should sense the output of the circuit and should not fire $b*$ — an output from its point of view — if $a*$ occurs. This can result in some anomalous behavior in the part of the environment that generates signal b. However, from the circuit side there is no problem, the output is always issued. Thus, speed-independence is possible from the circuit side. Eventually, a warning can be issued stating that there can be problems from the environment side.

Synthesizing a circuit for the behavior in figure 4.11, assuming x and z as inputs and y as output, we get an implementation consisting of a single wire connecting input x to output y.

Finally, let us consider a as an input and b as an output. This is the reverse of the previous case, thus, there are no problems from the environment side, but it can exist anomalous behavior in the circuit output. A change in signal b can eventually be forced to move backwards, because of the occurrence of input signal transition $a*$.

Figure 4.12.d shows a non-speed independent circuit implementation of the behavior depicted in figure 4.11, assuming x and y as inputs and z as output. The analysis of this circuit must be seen in the light of the behavior it implements. Thus, signal transition $y+$ occurs always after signal

transition $x+$. If the occurrence of $y+$ is far from that of $x+$, the output of the upper NOR gate rises and the z output remains at 0, even after the occurrence of $y+$. However, if the occurrence of $y+$ is too close to the occurrence of $x+$, not allowing the circuit to settle in response to this last event, the circuit can have an anomalous behavior at output z. Figure 4.12.e shows a possible speed-independent implementation for the same specification.

4.2.3 Non-Persistencies, Conclusions

From the previous study on symmetric and asymmetric non-persistencies we can conclude the following about this type of concurrent conflicts:

1. Whenever an input signal transition is disabled by another input signal transition, there is no speed constraint on the circuit side. It is up to the environment to guarantee the correct occurrence of the events and it is quite acceptable to assume the environment behaves correctly.

2. Whenever an input signal transition is disabled by an output signal transition, there is the possibility of wrong behavior in the input signal. The input signal is an output of the environment, which eventually has to stop its change because of the change in the circuit output signal.

3. Whenever an output signal transition is disabled by any other signal transition, there is the possibility of anomalous behavior on the output signal, which can assume the form of metastability or oscillation. A speed-independent, pure digital circuit cannot be defined to implement this signal. However, a speed-independent solution, based on special, partially analog devices, like the mutual exclusion element, can be feasible.

A symmetric or asymmetric non-persistency is an *SI conflict* if it contains a disabling of an output signal transition by any other signal transition. If the non-persistency contains a disabling of an input signal transition by an output signal transition, it is potentially an SI conflict on the environment side. Thus, a warning must be reported to the designer.

4.3 Managing Non-Persistencies

In the previous section we have analyzed non-persistency in state-graph specifications and we have pointed out that partially analog devices can be used to obtain a speed-independent implementation of a specification with speed-independent conflicts. Now, we will evolve in order to define a systematic approach to obtain a speed-independent solution, at least for a class of specifications. The approach is based on the theory of regions.

Let us start by introducing the notions of concurrency and mutual exclusivity between regions. Two events are *concurrent* if there is a state where they are simultaneously enabled. Two regions are said to be *concurrent* if their input interfaces are disjoint and concurrent, that is, every event from the input interface of one of the regions is concurrent to every event from the input interface of the other region. Two regions are said to be *mutually exclusive* if they have no common states.

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph and $R_1 = \langle I_1, O_1 \rangle$ and $R_2 = \langle I_2, O_2 \rangle$ two regions, defined by their interfaces.

Definition 4.4 (concurrent regions)
R_1 *and* R_2 *are said to be* concurrent *if*

1. $I_1 \cap I_2 = \emptyset$*; and*

2. $\forall_{e_1 \in I_1}, \forall_{e_2 \in I_2} : e_1$ *and* e_2 *are concurrent.*

Definition 4.5 (mutual exclusive regions)
R_1 *and* R_2 *are said to be* mutual exclusive *if* $R_1 \cap R_2 = \emptyset$*.*

In the last definition, R_1 and R_2 are used to represent the sets of states of the regions, instead of their interfaces. This procedure will be sometimes taken in sequel as long as no confusion arises.

Let's illustrate the notions of concurrency and mutual exclusivity between regions with an example.

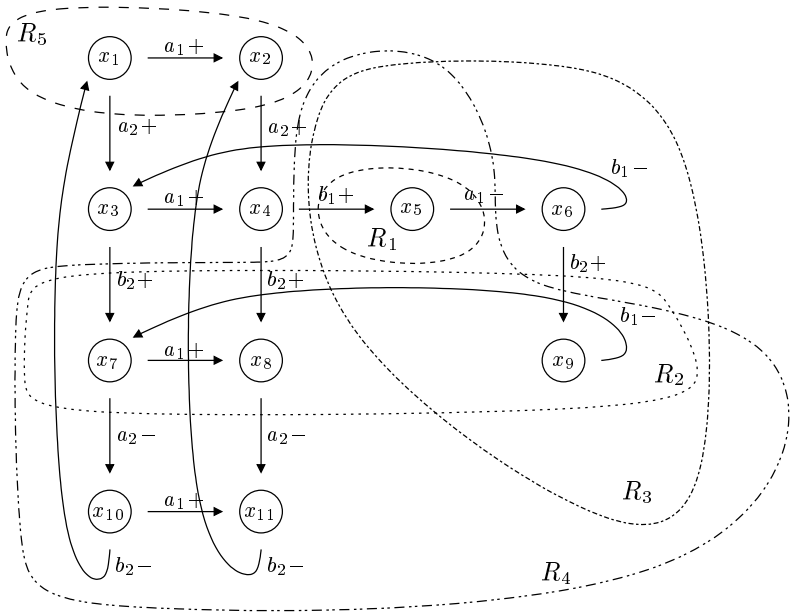


Figure 4.13: A state graph with 5 regions marked. Regions R_3 and R_4 are neither concurrent nor mutual exclusive; regions R_3 and R_5 are mutual exclusive, but not concurrent; regions R_2 and R_3 are concurrent, but not mutual exclusive; finally, regions R_1 and R_2 are concurrent and mutual exclusive.

Figure 4.13 shows a state graph, where 5 regions are signaled. They are

$$
\begin{aligned}
R_1 &= \langle \{b_1+\}, \{a_1-\} \rangle, \\
R_2 &= \langle \{b_2+\}, \{a_2-\} \rangle, \\
R_3 &= \langle \{b_1+\}, \{b_1-\} \rangle, \\
R_4 &= \langle \{b_1+, b_2+\}, \{a_1-, b_2-\} \rangle, \\
R_5 &= \langle \{b_2-\}, \{a_2+\} \rangle
\end{aligned}
$$

Regions R_3 and R_4 are neither concurrent, nor mutual exclusive: event b_1+ is common to the input interfaces of both regions and states x_5 and x_9 are common to the two regions. Regions R_3 and R_5 are mutual exclusive, but not concurrent: events b_1+ and b_2- are not concurrent. Regions R_2 and R_3 are concurrent, but not mutual exclusive: state x_9 is common to both regions. Finally, regions R_1 and R_2 are concurrent and mutual exclusive.

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph and $R_1 = \langle I_1, O_1 \rangle$ and $R_2 = \langle I_2, O_2 \rangle$ two regions, defined by their interfaces. Let define a morphism $m = \langle \sigma, \eta \rangle$ from G into a transition system $G' = \langle S', E', \Theta' \rangle$

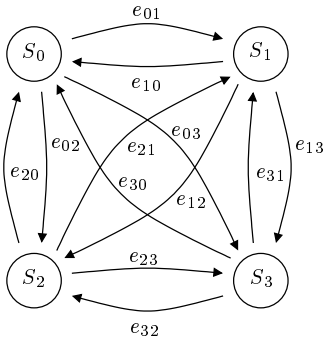


Figure 4.14: General regions collapse by a pair of regions.

defined as follows:

$$S' = \{S_0, S_1, S_2, S_3\}$$
$$E' = \{e_{01}, e_{02}, e_{03}, e_{10}, e_{12}, e_{13}, e_{20}, e_{21}, e_{23}, e_{30}, e_{31}, e_{32}\}$$
$$\Theta' = \{\langle S_i, e_{ij}, S_j\rangle | i, j \in \{0, 1, 2, 3\} \wedge i \neq j\}$$

The initial state of G' is left undefined, as it depends on the initial state of G. Transition system G' is depicted in figure 4.14 and represents a complete graph of order 4.

The mapping on states is the function $\sigma : S \mapsto S'$, such that

$$S_0 = S - R_1 - R_2, \quad S_1 = R_1 - R_2$$
$$S_2 = R_2 - R_1 \quad , \quad S_3 = R_1 \cap R_2$$

In these definitions, R_1 and R_2 represent the sets of states of the regions. The mapping on events is the function $\eta : E \mapsto E'$, such that

$$e_{ij} = \{e \in I_1 \cup I_2 \cup O_1 \cup O_2 \mid \quad \exists \langle s, e, s'\rangle \in \Theta : \sigma(s) = S_i \wedge \sigma(s') = S_j\}$$

Henceforth we will call this morphism the *regions collapse of G by the pair of regions $\langle R_1, R_2\rangle$*. Note that this morphism is not exactly the same as the projection of G by the set of events of the regions interfaces, as defined in section 3.2, although they have the same information. On the regions collapse side, some of the states as well as some of the transitions can be empty sets. On the projection side, an ordered pair of states can be connected by a multi-arc, instead of a single arc. Removing, in

the former, the empty states and the empty transitions and merging, in the latter, the multi-arcs to a single arc, with a label equal to the sets of labels of the multi-arcs, the two representations become isomorphic.

Theorem 4.6 (concurrent regions)

If regions R_1 and R_2 are concurrent, then $I_1 \cap O_2$ and $I_2 \cap O_1$ are empty sets.

Proof

The proof is done by contradiction. Assume on the contrary that $I_1 \cap O_2$ is not the empty set. Then, there is an event $e_1 \in I_1 \cap O_2$ and every state where e_1 is enabled belongs both to R_2 and to $S - R_1$. By condition 2 of definition 4.4, for every event $e_2 \in I_2$, there is a state $s \in S$, where e_1 and e_2 are simultaneously enabled. Such a state s belongs to both $S - R_1$ and $S - R_2$. But we already know that every state where e_1 is enabled belongs to R_2, which leads to a contradiction. So, $I_1 \cap O_2$ is the empty set. Similarly, we can prove $I_2 \cap O_1$ is also the empty set.

Let us give some examples of regions collapses. Let G be the state graph in figure 4.13. The regions collapses of G by the pairs of regions $\langle R_3, R_4 \rangle$, $\langle R_3, R_5 \rangle$, $\langle R_2, R_3 \rangle$ and $\langle R_1, R_2 \rangle$ are depicted in figure 4.15. Empty sets of events have been omitted in the picture. Cases (b) and (d) have $S_3 = \emptyset$, because the collapse is determined by mutual exclusive regions. Cases (c) and (d) have concurrent regions. As a consequence events e_{03}, e_{12} and e_{21} are the empty set. By condition 1 of definition 4.4, $I_1 \cap I_2 = \emptyset$. Then, there is no transition $\langle s, e, s' \rangle \in \Theta$, such that $s \in S - R_1 - R_2$ and $s' \in R_1 \cap R_2$ and hence e_{03} is the empty set. By theorem 4.6 $I_1 \cap O_2 = \emptyset$. Then, there is no transition $\langle s, e, s' \rangle \in \Theta$, such that $s \in R_2 - R_1$ and $s' \in R_1 - R_2$ and hence e_{12} is the empty set. Similarly, e_{21} also is the empty set. However, the reverse is not true: case (b) have empty e_{03}, e_{12} and e_{21} events, but regions R_3 and R_5 are not concurrent.

The definition of concurrency between regions does not determine event e_{30} to be the empty set. That's because $O_1 \cap O_2$ can be different from the empty set. In the state graph of figure 4.16.a, regions R_1 and R_2 are concurrent, but $O_1 \cap O_2 = \{c+\}$. As a consequence, the regions collapse of the state graph by regions R_1 and R_2 has a non-empty e_{30} event. Actually, $e_{30} = O_1 \cap O_2$.

Theorem 4.7 (concurrent and mutual exclusive regions)

If regions R_1 and R_2 are concurrent and mutual exclusive, then events e_{01}, e_{10}, e_{02}, e_{20}, and only them, are non-empty sets.

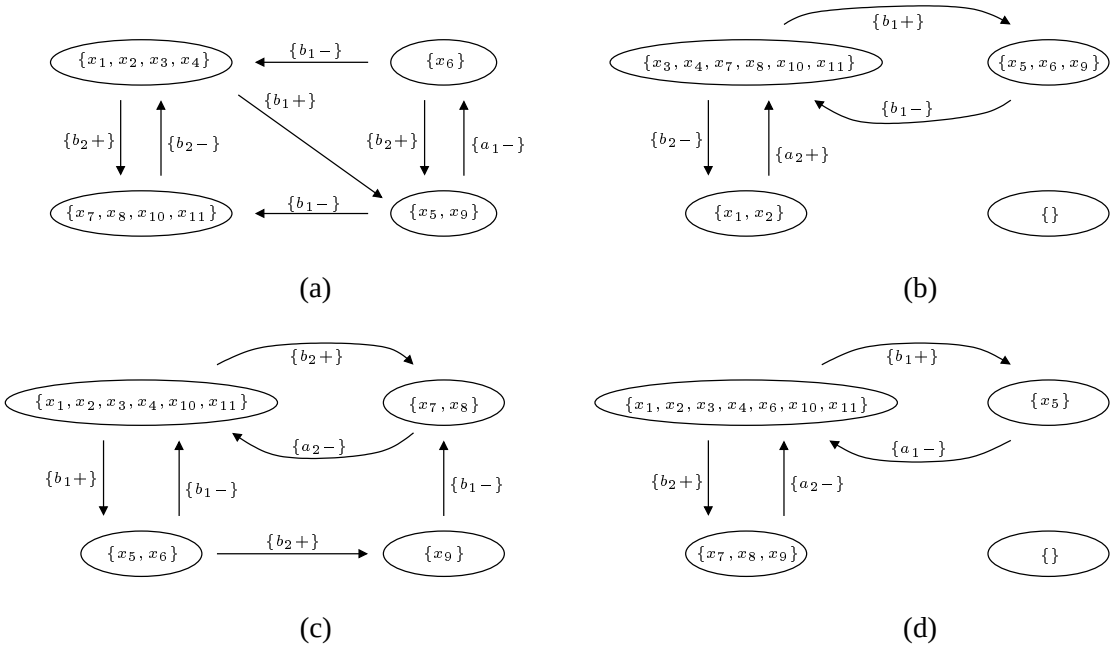


Figure 4.15: Regions collapses of the state graph in figure 4.13 by the pairs of regions (a) $\langle R_3, R_4 \rangle$, (b) $\langle R_3, R_5 \rangle$, (c) $\langle R_2, R_3 \rangle$ and (d) $\langle R_1, R_2 \rangle$.

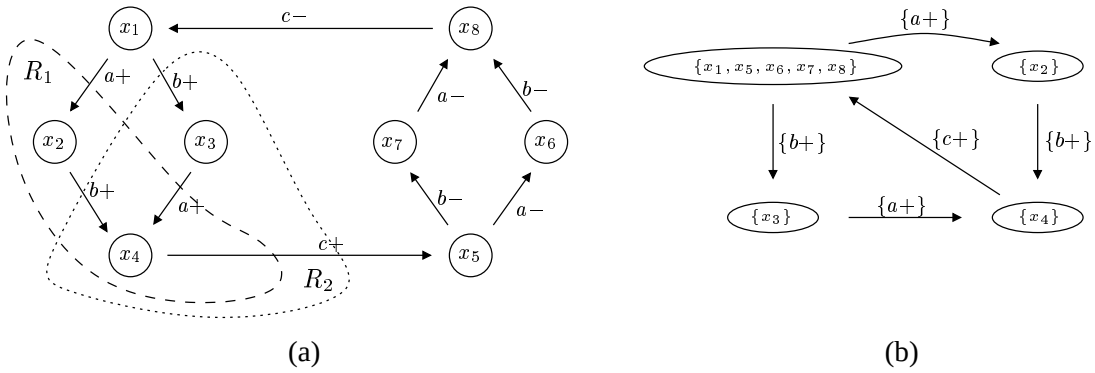


Figure 4.16: (a) The state graph has regions R_1 and R_2 concurrent, with common output interfaces. (b) Its regions collapse has a non-empty event e_{30}.

Proof

By theorem 4.6, sets of events e_{03}, e_{12} and e_{21} are empty sets. By definition 4.5, set $S_3 = R_1 \cap R_2 = \emptyset$. Then, sets of events e_{03}, e_{30}, e_{13}, e_{31}, e_{23} and e_{32} are empty sets. Thus, only e_{01}, e_{10}, e_{02}, e_{20} can eventually be non-empty sets. It is lacking to prove that they are necessarily non-empty sets. But, this is straightforward, since I_1, I_2, O_1 and O_2 are non-empty sets.

4.3.1 Mutex Insertion

The existence of a pair of concurrent and mutual exclusive regions on a state graph, determines the existence of a symmetric non-persistence between the events of the input interfaces. We know, from section 4.2, that if these events are transitions on output signals, some anomalous behavior can occur in a pure digital implementation of the state graph specification. We can understand this potential anomaly as a consequence of the impossibility to implement a fair mutual exclusion between the two concurrent regions. Thus, we delegate arbitration for the mutual exclusion to an external entity.

Let us have an arbitration device capable of realizing a fair mutual exclusion. As mentioned in chapter 1, the two-channel mutual-exclusion element (mutex) is an example of such a device. A cycle in channel i is a sequence of 4 events: a *request* (r_i+) to the mutex, a *grant* (g_i+) from the mutex, a *release* (r_i-) to the mutex and a *completion* (g_i-) from the mutex. The behavior of the mutex guarantees that only one grant is given at a time. A grant to a pending request is only given when the active cycle has completed.

We can modify our state graph in order to put the mutex controlling the access to the concurrent, mutual exclusive regions. Each region will be controlled by a different channel of the mutex. The control of each region $R_i = \langle I_i, O_i \rangle$ is done as follows. The request r_i+ is issued when the system is ready to enter the region. We will associate this with the minimum region having I_i as the output interface. The input interface events are delayed until the grant event g_i+ is received from the mutex. On the other side, the grant event should only appear after the request event has occurred.

Once the system has entered one of the exclusive regions, it remains inside it until the region is left. No special care must be taken in the insertions of the release and completion events of the external arbiter, as long as they are not done after the end of the region. A region having I_i as the input interface is a good candidate as a basis for the insertions. A region having O_i as the input interface

also is a good candidate. Let μ_i be a set of events that transversely cut region $\langle I_i, O_i \rangle$, that is, μ_i is such that $\langle I_i, \mu_i \rangle$ and $\langle \mu_i, O_i \rangle$ are regions. A region having μ_i as the input interface also is a good candidate as a basis for the insertions of the release and completion events. Thus, we can choose a good (ideally the best) region to base the insertions on, in order to achieve some results, like, for instance, minimize circuitry.

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph and $R_1 = \langle I_1, O_1 \rangle$ and $R_2 = \langle I_2, O_2 \rangle$ two regions, defined by their interfaces, such that, R_1 and R_2 are concurrent and mutually exclusive. Let M be a mutex and r_1, r_2, g_1 and g_2 be respectively its two input and two output lines. The insertion of channel i of mutex M to control access to regions R_i is determined by the following procedure:

Procedure 4.8 (insertion of a mutex channel)

1. *Let $\langle \alpha_i, I_i \rangle$ be the minimum region having I_i as the output interface. Insert r_i+ in G such that it appears after the α_i but before the I_i events.*

2. *Insert g_i+ in G such that it appears after the r_i+ event but before the I_i events.*

3. *Let $\langle \mu_i, \nu_i \rangle$ be the region used to insert the release and completion events. Insert r_i- such that it appears after the μ_i events but before the ν_i events.*

4. *Finally, insert g_i- such that it appears after the r_i- event but before the ν_i events.*[1]

Let us illustrate this approach with the specification given by the state graph depicted in figure 4.13 and repeated in figure 4.17. There is a symmetric non-persistency relation between events b_1+ and b_2+ in state x_4. Regions $R_1 = \langle \{b_1+\}, \{a_1-\} \rangle$ and $R_2 = \langle \{b_2+\}, \{b_2-\} \rangle$ are concurrent, mutual exclusive and their input interfaces coincide with the non-persistency events. Let now modify this state graph in order to control the access to regions R_1 and R_2 by means of a two-channel mutex. Region R_1 will be controlled by channel 1 of the mutex and region R_2 by channel 2. Let r_1 and r_2 be the inputs of the mutex, and g_1 and g_2 the outputs.

Let us start with region R_2. The minimal pre-region for event b_2+ has as input interface $\alpha_2 = \{a_2+\}$. The insertion of the release and completion events will be done based on region entered by a_2- and exited by b_2-. Thus, $\mu_2 = \{a_2-\}$ and $\nu_2 = \{b_2-\}$. Now, let's go to region R_1. The minimal pre-region for event b_1+ has as input interface $\alpha_1 = \{a_1+\}$. For the insertion of the release and

[1] For a detailed explanation of the transformation see chapter 5.

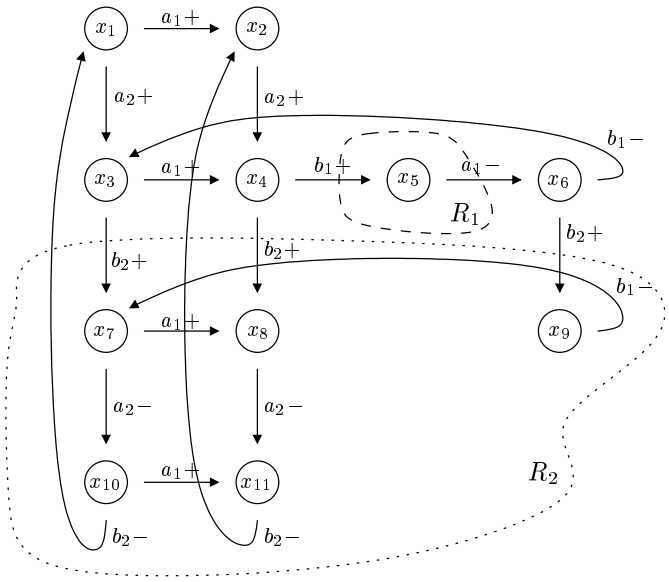


Figure 4.17: A state graph with a symmetric non-persistency between events b_1+ and b_2+. Regions R_1 and R_2 are concurrent, mutual exclusive and their input interfaces coincide with the non-persistency events. They can be used to control the non-persistency by means of a two-channel mutex.

completion events we choose region $\langle \mu_1, \nu_1 \rangle = \langle \{a_1-\}, \{b_1-\} \rangle$. The insertions result in the state graph depicted in figure 4.18. In this transformed state graph, the non-persistency between events b_1+ and b_2+ remains. This is expected because the transformations are persistency preserving.

However, we should remember that the g_1 and g_2 signals come from the mutex, which guarantee they are never at 1 at the same time. Thus, all states corresponding to g_1 and g_2 at 1 can be removed from the state graph. The complete transformation of a state graph in order to insert a mutex for controlling access to a pair of concurrent and mutual exclusive regions is thus given by the following procedure:

Procedure 4.9 (mutex insertion)

1. *Insert channel 1 of the mutex as determined by procedure 4.8.*
2. *Insert channel 2 of the mutex as determined by procedure 4.8.*
3. *Remove unreachable states, due to mutex behavior.*

For the example under study (figure 4.18), the states that can be removed are those encircled by the dashed line in the figure. After the removal the original non-persistency between output signal

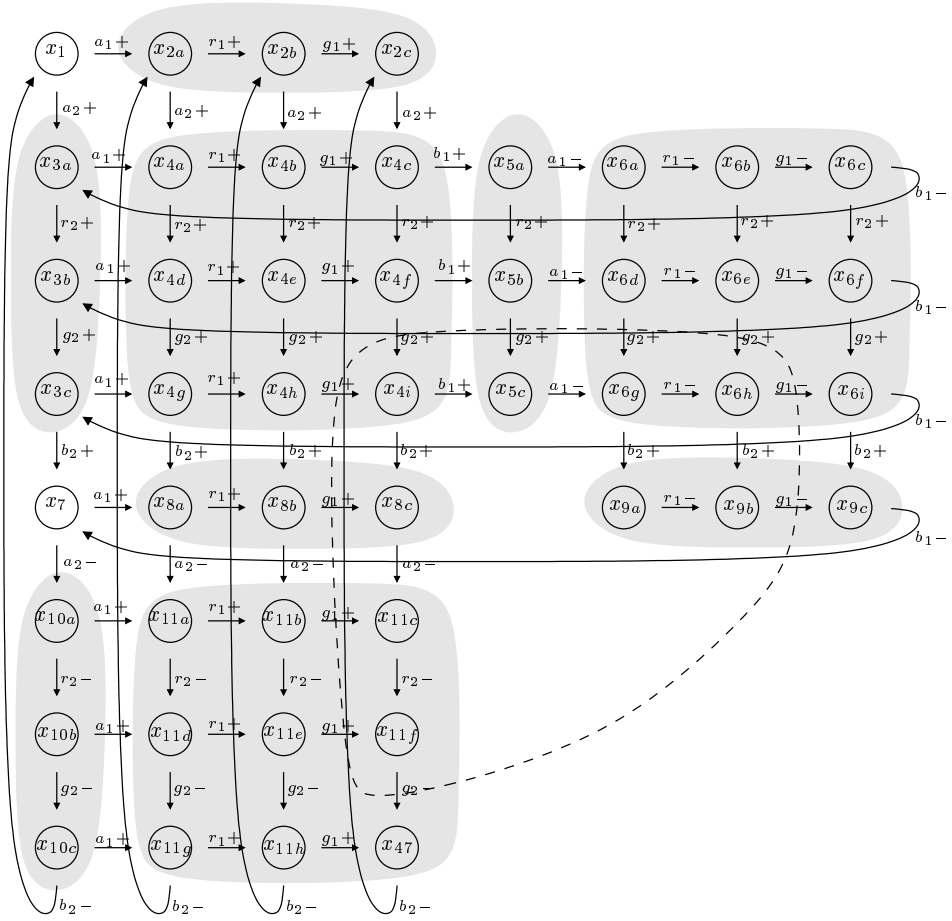


Figure 4.18: Transformed state graph obtained from the one in figure 4.17 after insertion of a mutex to control access to the concurrent and mutual exclusive regions. In order to highlight evolution from the original state graph, sets of states corresponding to the same original state are correlated by their names and by a common shadow background. States encircled by the dashed line can be removed, since, because of the mutex behavior, they are unreachable.

transitions b_1+ and b_2+ is transformed into a non-persistency between g_1+ and g_2+, two input signal transitions. Thus, the transformed specification can be synthesized as a speed-independent pure digital circuit. Applying the transformed state graph, with the unreachable states removed, to *petrify* produces the following equations to the output signals: $r_1 = a_1$, $r_2 = a_2$, $b_1 = g_1.a_2$ and $b_2 = g_2$. Hence, the total circuit is composed of a mutex plus an AND-gate.

Sometimes the excitation regions of the input interface events of the two concurrent, mutual exclusive regions are the same, that is, conditions for entering both regions are the same. In such cases, instead of the previous arbitration device, we use one with only one input, but two output lines. In response to a request in the unique input line, the arbiter must non-deterministically return a grant in one and only one of the output lines. A mutex with the two input lines short circuited is an example of such a device. (Refer back to figure 4.9.b to the state graph description of a mutex with the inputs short-circuited.) This device has only one request event, only one release event, but two grant and two completion events.

In this situation the insertion of the mutex to control access to the concurrent and mutual exclusive regions is done as follows. The request $r+$ is issued when the system is ready to enter both regions. We will associate this with the minimum region having $I_1 \cup I_2$ as the input interface. The grant events cannot be inserted based on $I_1 \cup I_2$, because each grant event must only delay events of the input interface of the region, whose access it is controlling. We will base insertions of the grant event $g+$ on the minimum pre-region of events I_i. Using different mechanisms for the insertions of the request and grant events will eventually result in some grant events occurring before the corresponding request events. This is not a problem because, due to the mutex behavior, they will be taken off of the specification.

Since the request event $r+$ is inserted based on both regions, the same must be done for the insertion of the release event $r-$. Similarly to what has been explained for the insertion of a two input, two output mutex, any region whose input interface transversely cut region $\langle I_1 \cup I_2, O_1 \cup O_2 \rangle$ is a good candidate.[2] Being the complement of the grant event g_i+, the completion event g_i- is specific for each region R_i. Let $\mu_1 \cup \mu_2$ be the input interface of the region used to insert the common release event, with μ_1 associated to R_1 and μ_2 to R_2. A region having μ_i as the input interface is a good candidate as a basis for the insertion of the completion event g_1-.

[2]Regions $\langle I_1, O_1 \rangle$ and $\langle I_2, O_2 \rangle$ are mutually exclusive. Thus, $\langle I_1 \cup I_2, O_1 \cup O_2 \rangle$ is a region.

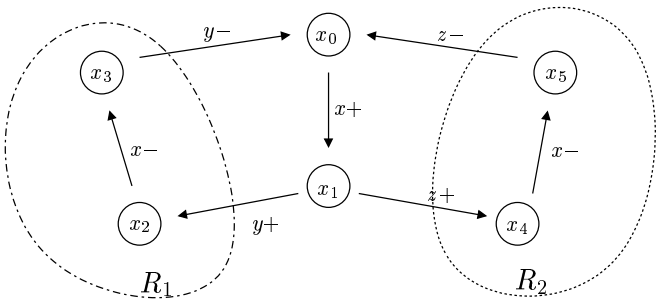


Figure 4.19: A state graph with a symmetric non-persistency between events $y+$ and $z+$. Regions R_1 and R_2 are concurrent, mutual exclusive and the excitation regions for their input interfaces coincide. The control of access to these regions can be done with a mutex with the inputs short circuited.

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph and $R_1 = \langle I_1, O_1 \rangle$ and $R_2 = \langle I_2, O_2 \rangle$ two regions, defined by their interfaces, such that, R_1 and R_2 are concurrent, mutual exclusive and the excitation regions for I_1 and I_2 are the same. In the state graph in figure 4.19, the two encircled regions are in these conditions. Let M be a mutex with the two inputs short circuited and let r, g_1 and g_2 be respectively its input and two output lines. The insertion of M in G to control access to regions R_1 and R_2 is determined by the following procedure:

Procedure 4.10 (mutex insertion)

1. *Let $\langle \alpha', I_1 \cup I_2 \rangle$ be the minimum region having $I_1 \cup I_2$ as the output interface. Insert $r+$ in G, such that, it appears after the α' events but before the $I_1 \cup I_2$ events.*

2. *Let $\langle \alpha_i, I_i \rangle$, for $i = 1, 2$, be the minimum region having I_i as the output interface. Insert g_i+ in G such that it appears after the α_i but before the I_i events.*

3. *Let $\langle \mu_1 \cup \mu_2, \nu' \rangle$ be the region chosen to base insertion of the release event on. Insert $r-$ in G, such that, it appears after the $\mu_1 \cup \mu_2$ events but before the ν' events.*

4. *Let $\langle \mu_i, \nu_i \rangle$, for $i = 1, 2$, be the regions chosen to insert the completion events. Insert g_i- in G such that it appears after the μ_i but before the ν_i events.*

5. *Remove unreachable states, due to mutex behavior.*

Let us now modify the state graph depicted in figure 4.19 in order to control access to regions R_1 and R_2 using a mutex M with the inputs short-circuited. Let r be the common input and g_1 and

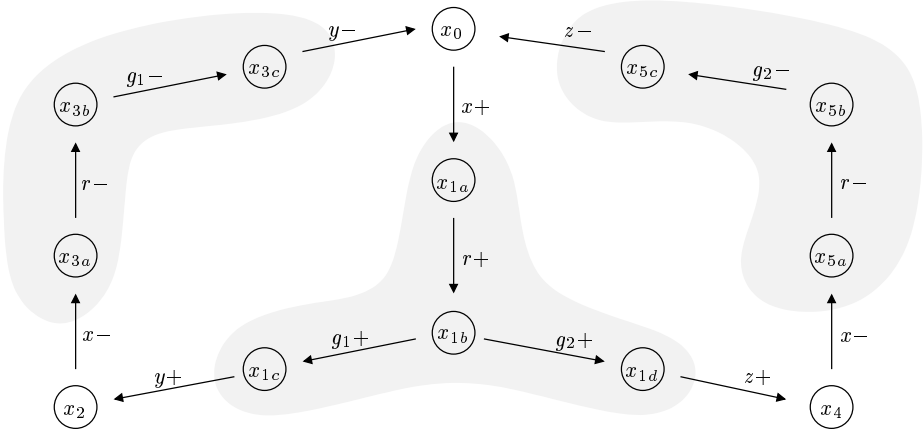


Figure 4.20: State graph obtained after transforming the state graph in figure 4.19, in order to control the access to regions R_1 and R_2 by means of a mutex with the inputs short circuited. Sets of states corresponding to the same original state are signaled by their names and by a common shadow background.

g_2 the two outputs. Event $r+$ is inserted based on region $\langle\{x+\},\{y+,z+\}\rangle$; event g_1+ is inserted based on region $\langle\{y-\},\{y+\}\rangle$; event g_2+ is inserted based on region $\langle\{z-\},\{z+\}\rangle$; event $r-$ is inserted based on region $\langle\{x-\},\{y-,z-\}\rangle$; event g_1- is inserted based on region $\langle\{y+\},\{y-\}\rangle$; and event g_2- is inserted based on region $\langle\{z+\},\{z-\}\rangle$. Finally, the overall behavior is conditioned by the behavior of the mutex. The overall result is the state graph depicted in figure 4.20. The original state graph, with a symmetric non-persistency between two output signal transitions, which represents a SI conflict, was transformed into a state graph with a symmetric non-persistency between two input signal transitions, which are the outputs of the mutex. There is no SI conflict in the latter specification and so an implementation can be generated using a speed-independent synthesis tool. This specification can be synthesized to

$$r = x$$
$$y = g1$$
$$z = g2,$$

plus the mutex, which corresponds to the implementation we have proposed in section 4.2.1.

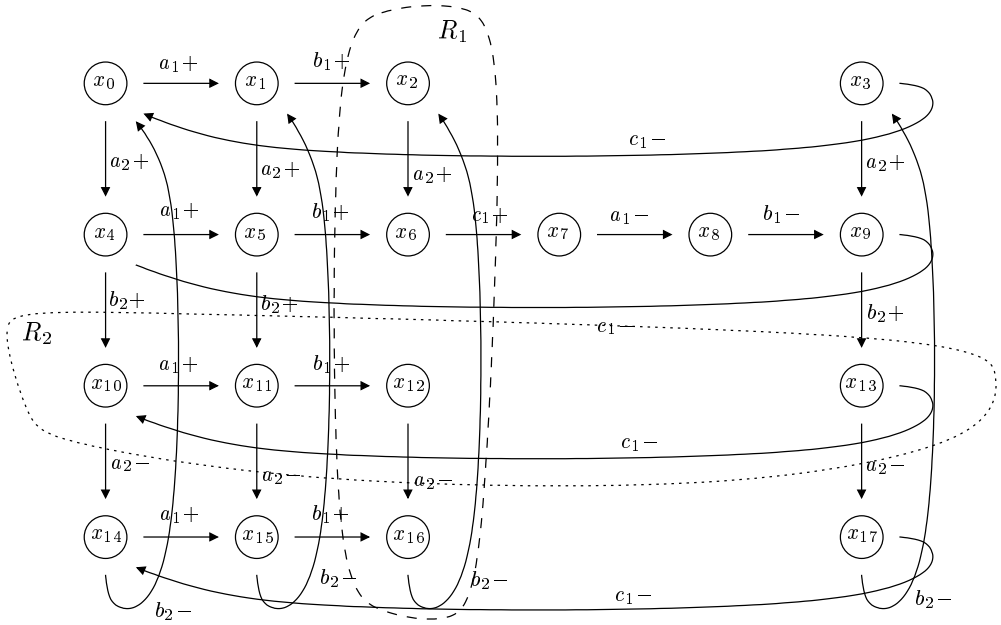


Figure 4.21: A state graph with an asymmetric non-persistency between signal transitions b_1+ and b_2+, b_2+ being disabled by b_1+. Regions R_1 and R_2 are the minimal regions having respectively b_1+ and b_2+ as the input interface. Regions R_1 and R_2 are concurrent, but they are not mutual exclusive: state x_{12} belongs to both regions.

4.3.2 Empty Regions

The previous examples point out a possible solution to manage a class of SI conflicts, those assuming the form of non-persistencies. Given the non-persistency, we identify two concurrent, mutually exclusive regions having the non-persistent events as the input interfaces. Then, controlling access to these regions by means of a mutex, we transfer the SI conflict to inside the mutex, getting a specification free from the SI conflict.

However, it is not always possible to find out two concurrent, mutual exclusive regions covering the conflict. Consider, for instance, the state graph depicted in figure 4.21, where a_1 and a_2 are input signals and b_1, c_1 and b_2 are output signals. There is an asymmetric non-persistency between signal transitions b_1+ and b_2+, with b_2+ being disabled by b_1+. The minimal concurrent regions having b_1+ and b_2+ as input interfaces are $R_1 = \langle\{b_1+\},\{c_1+\}\rangle$ and $R_2 = \langle\{b_2+\},\{a_2-\}\rangle$ and are signaled in the figure. They are not mutual exclusive. Controlling access to these regions using a

mutex would result in state x_{12} becoming unreachable.

A solution can eventually be found extending the notion of region. Region has been defined as a set of states with a fixed entry/exit relation. There is a set of events that enter the region and a set of events that exit it. There is no event common to the two sets: if an event enters the region it cannot leave it. Let us extend the definition of region such that we include a special one, empty of states, which is entered and exited by the same set of events. Such an *empty region* is only definable by its interface, since it does not contains any internal states. For instance, in the state graph of figure 4.21, we can define the empty region entered and exited by event b_2+, denoting it by $R_3 = \langle\{b_2+\}, \{b_2+\}\rangle$.

According to definition 4.4, regions R_1, defined above, and R_3 are concurrent. But are they mutual exclusive? The notion of mutual exclusion between regions must be seen with some care. For non-empty regions we have relied on the regions intersection. If there is no common state, the regions are mutually exclusive. But this can not be applied with empty regions: the intersection in terms of states is always the empty set. Consider empty region $R_4 = \langle\{b_1+\}, \{b_1+\}\rangle$ and its intersection with region R_2, defined above, which is concurrent to R_4. There is no common state, but transition $\langle x_{11}, b_1+, x_{12}\rangle$ is internal to region R_2. Thus, they are not mutual exclusive.

The definition of mutual exclusion between regions (definition 4.5) can be extended in order to cover empty regions.

Definition 4.11 (mutual exclusive regions)
Let G be a state graph, R a non-empty region and R' another (eventually empty) region. R and R' are said to be mutual exclusive *if no transition from R' is internal to R.*

We do not consider mutual exclusion between empty regions. If R and R' are non-empty regions and $R \cap R' \neq \emptyset$, then there is a transition from R' internal to R. Thus, definition 4.11 is indeed an extension of definition 4.5.

Regions R_3 and R_1 are mutual exclusive, and thus, they can eventually be used to control the non-persistency. Also regions R_3 and $R_5 = \langle\{b_1+\}, \{a_1-\}\rangle$ are mutual exclusive, as well as regions R_3 and $R_6 = \langle\{b_1+\}, \{b_1-\}\rangle$. However, regions R_3 and $R_7 = \langle\{b_1+\}, \{c_1-\}\rangle$ are not mutual exclusive, since transition $\langle x_9, b_2+, x_{13}\rangle$ is internal to R_7.

We will use regions R_3 and R_6 to control the non-persistency. Since the excitation regions of the two input interfaces do not coincide, we apply procedures 4.8 and 4.9. For the insertion of channel

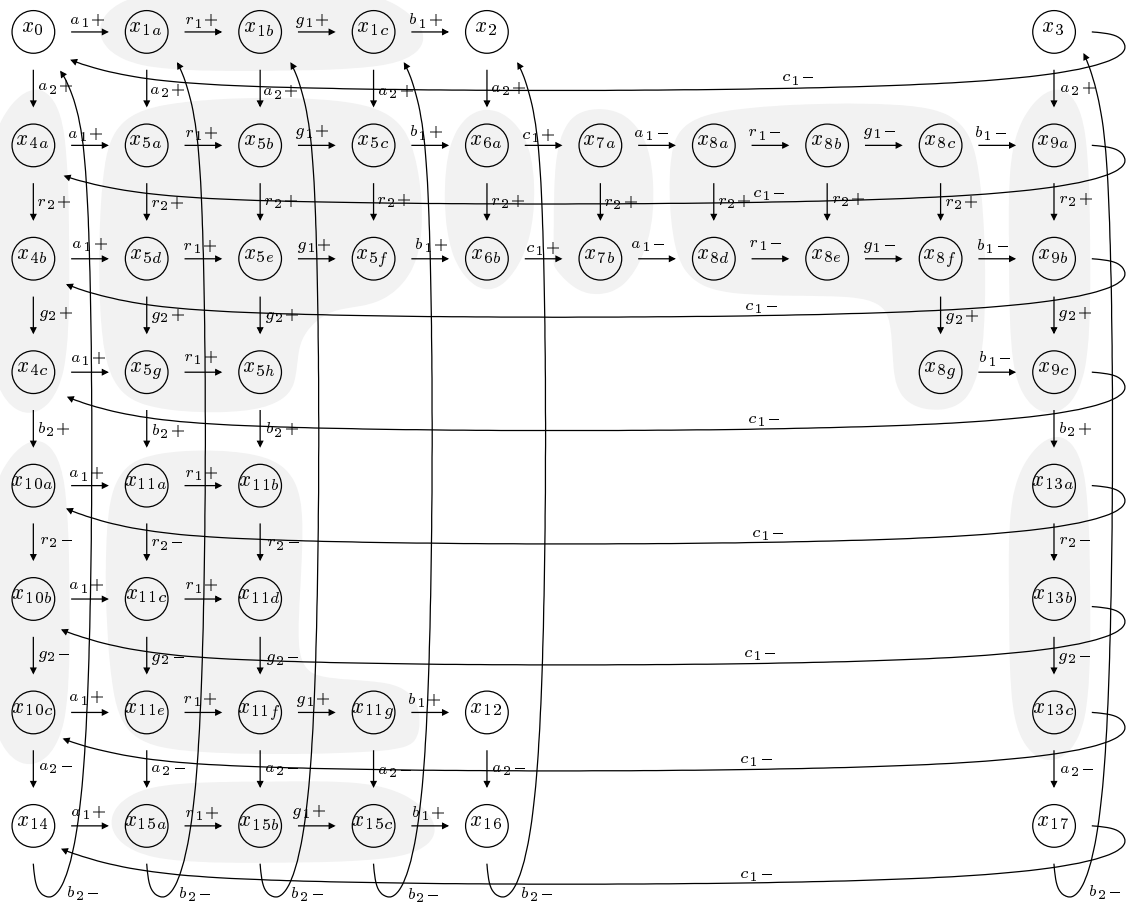


Figure 4.22: State graph obtained after transforming the state graph depicted in figure 4.21 in order to insert a mutex to control the non-persistency between output signal transitions b_1+ and b_2+. The two concurrent and mutual exclusive regions used as the basis for the mutex insertion were $R_6 = \langle\{b_1+\},\{b_1-\}\rangle$ and $R_3 = \langle\{b_2+\},\{b_2+\}\rangle$, the latter being an empty region. Sets of states corresponding to the same original state are signaled by their names and by a common shadow background.

1 we use $\alpha_1 = \{a_1+\}$, $\mu_1 = \{a_1-\}$ and $\nu_1 = \{b_1-\}$. For the insertion of channel 2 we use $\alpha_2 = \{a_2+\}$, $\mu_2 = \{b_2+\}$ and $\nu_2 = \{a_2-\}$. After applying procedure 4.9 the state graph in figure 4.21 is transformed into the state graph in figure 4.22. The asymmetric non-persistency between output signal transitions is transformed into a symmetric non-persistency between input signal transitions.

4.3.3 General Exclusion

So far the control of non-persistency SI conflicts rely on the acquisition of pairs of concurrent and mutual exclusive regions. Eventually, one of the regions can be an empty region. However, sometimes such a pair of concurrent and mutual exclusive regions cannot be found. Consider the circuit description given by the Petri net (STG) depicted in figure 4.23.a. Its partial reachability graph, including the interesting parts, is depicted in figure 4.23.b. In the initial marking, transitions b_1+, b_2+ and b_3+ are enabled, but there are only 2 tokens in place p. Thus, once one of these transitions fire, we have a symmetric non-persistency between the other two. In the (partial) state graph description we can see:

1. a symmetric non-persistency between events b_1+ and b_2+ in state 001.
2. a symmetric non-persistency between events b_1+ and b_3+ in state 010.
3. a symmetric non-persistency between events b_2+ and b_3+ in state 100.[3]

Assume that we want to control these non-persistencies using two input, two output mutexes. Consider, for instance, that we want to control the non-persistency between b_1+ and b_2+. The minimum regions having $\{b_1+\}$ and $\{b_2+\}$ as the input interfaces are respectively $R_1 = \langle\{b_1+\}, \{a_1-\}\rangle$ and $R_2 = \langle\{b_2+\}, \{a_2-\}\rangle$. These two regions are concurrent but not mutual exclusive. Even if we consider empty regions, no mutual exclusive pair of regions can be found. Indeed, transition $\langle 100, b_2+, 110\rangle$ is internal to region R_1 and transition $\langle 010, b_1+, 110\rangle$ is internal to region R_2. Thus, we cannot control the non-persistency between b_1+ and b_2+ using a two output mutex.

If we have chosen any one of the others two non-persistencies, the results will be quite similar. That's because the three non-persistencies are not independent each one from the others. The three non-persistencies are actually caused by an exclusion relation among regions, more general than the mutual exclusion we have considered so far. Looking at the Petri net depicted in figure 4.23.a, it is quite easy to accept that $R_1 = \langle\{b_1+, b_1-\}\rangle$, $R_2 = \langle\{b_2+, b_2-\}\rangle$ and $R_3 = \langle\{b_3+, a_3+\}\rangle$ are regions. These three regions are concurrent two by two. These three regions also dispute the two tokens in place p. The system can be simultaneously inside any two of these three regions, but not

[3] Actually, if we have drawn all the reachability graph, we could see that the number of states for each non-persistency was greater.

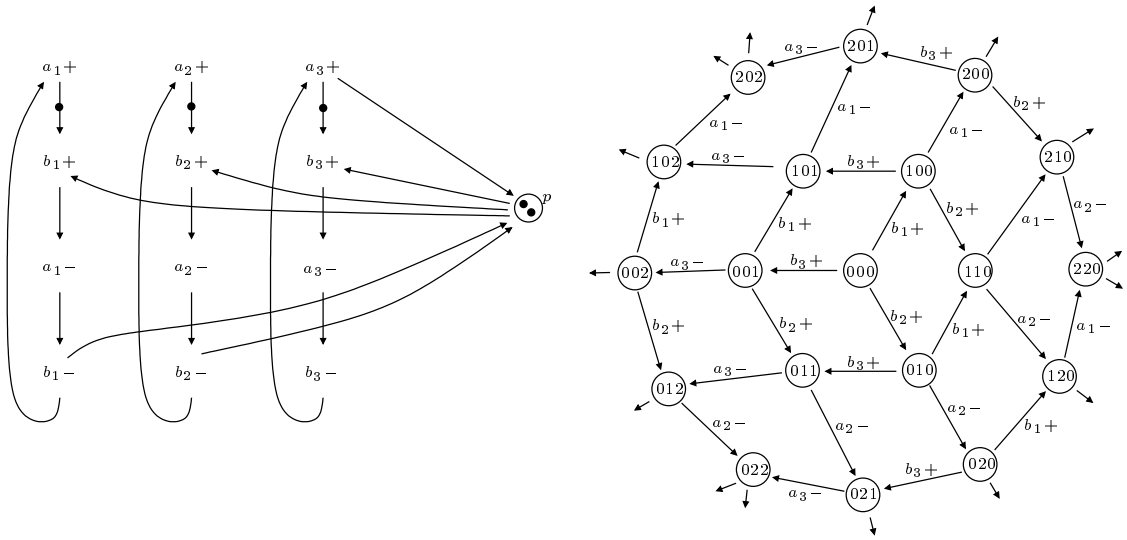


Figure 4.23: A circuit specification with symmetric non-persistency relations involving three signal transitions, namely b_1+, b_2+ and b_3+: (a) Petri net (STG) description; (b) partial state graph description, including states where the non-persistencies manifest themselves, like states 001, 010 and 100.

inside the three at the same time. Thus, the intersection of the three regions is the empty set. These three regions are in, what we call, an *exclusion relation of 2 out of 3*.

Let G be a state graph and R_1, R_2 and R_3 three regions in G.

Definition 4.12 (concurrency)

R_1, R_2 and R_3 are said to be concurrent *if they are concurrent two by two. In general, n different regions are said to be* concurrent *if they are concurrent two by two.*

Definition 4.13 (exclusion 2 out of 3)

R_1, R_2 and R_3 are said to be in an exclusion relation 2 out of 3 *if the following two conditions hold:*

1. $\forall_{i,j \in \{1,2,3\}} : i \neq j \Rightarrow R_i \cap R_j \neq \emptyset$

2. $R_1 \cap R_2 \cap R_3 = \emptyset$

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph and $R_1 = \langle I_1, O_1 \rangle$, $R_2 = \langle I_2, O_2 \rangle$ and $R_4 = \langle I_4, O_4 \rangle$ three regions, defined by their interfaces.[4] Let define a morphism $m = \langle \sigma, \eta \rangle$ from G into a transition system $G' = \langle S', E', \Theta' \rangle$ defined as follows:

$$S' = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}$$
$$E' = \{e_{ij} \mid i, j \in \{0, 1, \cdots, 7\} \wedge i \neq j\}$$
$$\Theta' = \{\langle S_i, e_{ij}, S_j \rangle \mid i, j \in \{0, 1, \cdots, 7\} \wedge i \neq j\}$$

The initial state of G' is left undefined, as it depends on the initial state of G. The mapping on states is the function $\sigma : S \mapsto S'$, such that

$$S_0 = S - R_1 - R_2 - R_4$$
$$S_1 = R_1 - R_2 - R_4 \quad S_2 = R_2 - R_1 - R_4 \quad S_4 = R_4 - R_1 - R_2$$
$$S_3 = (R_1 \cap R_2) - R_4 \quad S_5 = (R_1 \cap R_4) - R_2 \quad S_6 = (R_2 \cap R_4) - R_1$$
$$S_7 = R_1 \cap R_2 \cap R_4$$

In these definitions, R_1, R_2 and R_4 represent the sets of states of the regions. The mapping on events is the function $\eta : E \mapsto E'$, such that

$$e_{ij} = \{e \in I_1 \cup I_2 \cup I_4 \cup O_1 \cup O_2 \cup O_4 \mid \quad \exists \langle s, e, s' \rangle \in \Theta : \sigma(s) = S_i \wedge \sigma(s') = S_j\}$$

This morphism will be called the *regions collapse of G by the triple of regions $\langle R_1, R_2, R_3 \rangle$*.

Theorem 4.14 (exclusion 2 out of 3)

If regions R_1, R_2 and R_4 are concurrent and are in an exclusion relation of 2 out of 3, then all events other than e_{01}, e_{02}, e_{04}, e_{10}, e_{13}, e_{15}, e_{20}, e_{23}, e_{26}, e_{30}, e_{31}, e_{32}, e_{40}, e_{45}, e_{46}, e_{50}, e_{51}, e_{54}, e_{60}, e_{62} and e_{64} are empty sets.

Proof

By the definition of concurrency between regions (definition 4.4), $I_1 \cap I_2 = I_1 \cap I_4 = I_2 \cap I_4 = \emptyset$. Thus, e_{03}, e_{05} and e_{06} are the empty sets. By the definition of exclusion 2 out of 3 (definition 4.13), $R_1 \cap R_2 \cap R_4 = \emptyset$. Thus, e_{i7} and e_{7i}, for $i = 0, 1, \cdots, 6$, are the empty sets. Finally, by theorem 4.6, $I_i \cap O_j$, for $i \neq j$, is the empty set. Thus, e_{12}, e_{21}, e_{14}, e_{41}, e_{24}, e_{42}, e_{56}, e_{65}, e_{36}, e_{63}, e_{35}, e_{53}, e_{16}, e_{61}, e_{25}, e_{52}, e_{43} and e_{34} also are empty sets, which concludes the proof.

[4] The third region is called R_4 and not R_3 because it makes easier the indices used below. The indices 1, 2 and 4 stand for the binary combinations 001, 010 and 100 respectively.

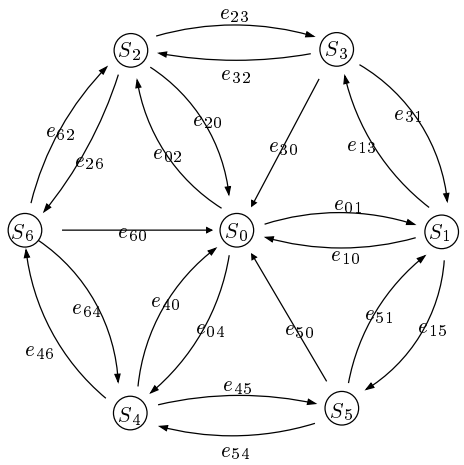
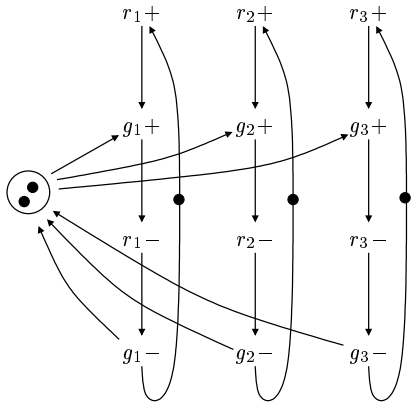


Figure 4.24: General form of the regions collapse of a state graph by a triple of regions, when they are concurrent and are in an exclusion relation of 2 out of 3.

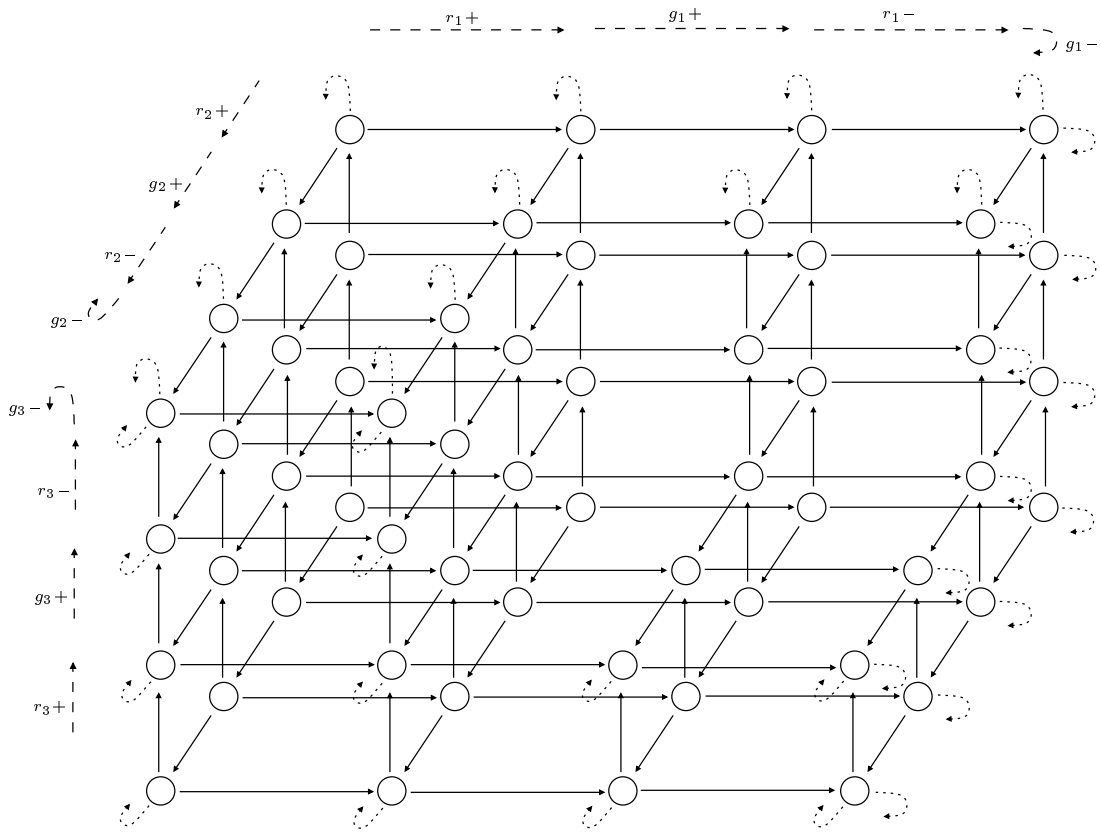
If regions R_1, R_2 and R_4 are concurrent and are in an exclusion relation of 2 out of 3, the regions collapse of G by the triple $\langle R_1, R_2, R_4 \rangle$, takes the general form represented in figure 4.24.

Let us have an arbitration device with 3 input and 3 output lines. It accepts requests on the input lines and delivers grants on the output lines. Its behavior is such that at most 2 grants are given at a time. If one or two requests are issued, they are automatically granted. But a third request must wait until a release/completion, before it is granted by the arbiter. This device realizes an exclusion of 2 out of 3. Let's call this arbiter a *genex 3x2*. The behavior of a genex 3x2 can be described by the Petri net depicted in figure 4.25.a and by the corresponding reachability (state) graph depicted in figure 4.25.b. For the purpose of this chapter we are not interested on how to build a genex 3x2. It is assumed as a black box, with 3 inputs and 3 outputs, whose behavior is the one previously defined. We will return to this device in another chapter.

The control of access to 3 concurrent regions, which are in an exclusion relation of 2 out of 3, can be done by means of a genex 3x2. We would expect that the insertion of each channel of a genex could be done the same way we have inserted channels of a mutex, that is, using procedure 4.8. However, this is not true. Let's explain why with the help of figure 4.26. In part (a) we are representing 3 regions of the same system, which we assume are concurrent and in an exclusion relation of 2 out of 3. For instance, we can associate these regions with regions $R_1 = \langle b_1+, b_1- \rangle$, $R_2 = \langle b_2+, b_2- \rangle$



(a)



(b)

Figure 4.25: Behavior of a genex 3x2: (a) Petri net description; (b) State graph description.

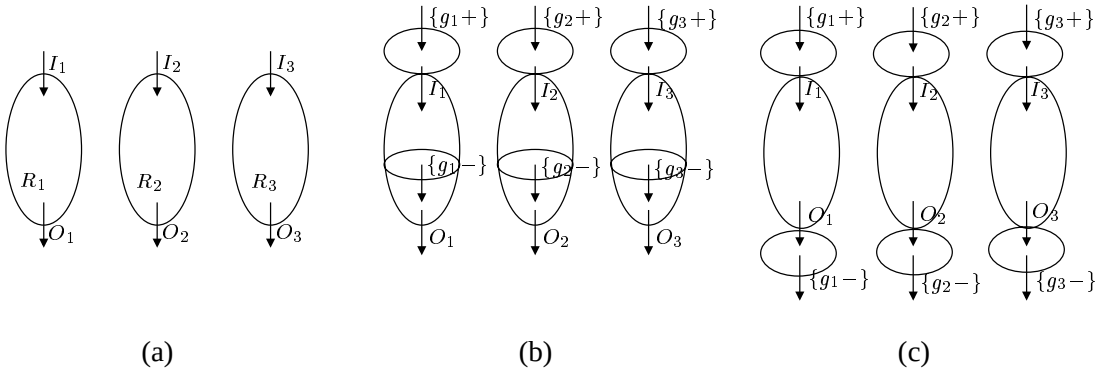


Figure 4.26: Representation of three concurrent regions of a given state graph, which are in an 2 out of 3 exclusion relation, and 2 different approaches to insert a genex 3x2 to control access to these regions. (a) original regions in an exclusion relation of 2 out of 3. (b) incorrect insertion of the genex 3x2: in this case the system can evolve to a state, for instance, simultaneously inside regions $\langle\{g_1\}, O_1\rangle$, $\langle\{g_2+\}, I_2\rangle$ and $\langle\{g_3+\}, I_3\rangle$, where the original non-persistencies persist. (c) correct insertion of the genex 3x2.

and $R_3 = \langle b_3+, a_3+\rangle$ of the system depicted in figure 4.23. These regions are not completelly visible in the partial state graph (figure 4.23.b), but the information can be extracted from the Petri net description (figure 4.23.a), considering the transitions that interact with place p. Consider that we control access to these regions using a genex 3x2, inserting the release-completion events before the end of the regions, as is illustrated in figure 4.26.b. Assume now that the system evolves to a state, which is inside region R_1, after the occurrence of event g_1-, and inside the excitation regions of events g_2+ and g_3+. Both events, g_2+ and g_3+ can occur, since it is allowed by the genex. Thus, the system can evolve to a state inside the excitation regions of events I_2 and I_3. But the system is still inside region R_1, so, in this last state, there is a non-persistency not removed by the genex. The release-completion events of the genex channels must be inserted at the end of the regions to be controlled. This is illustrated by figure 4.26.c.

Let $G = \langle S, E, \Theta, s_{in}\rangle$ be a state graph and $R_1 = \langle I_1, O_1\rangle$, $R_2 = \langle I_2, O_2\rangle$ and $R_3 = \langle I_3, O_3\rangle$ three regions, defined by their interfaces, which are concurrent and in an exclusion relation of 2 out of 3. Let M be a genex 3x2, being r_i and g_i, for $i \in \{1, 2, 3\}$, its input and output signals. The transformation of G in order to insert the genex 3x2 to control access to regions R_1, R_2 and R_3 is done as follows. Each region is controlled by a different channel of the genex. Let channel i control

access to region R_i. The request event, r_i+, is inserted based on the minimum region having I_i as the output interface. Let $\langle \alpha_i, I_i \rangle$ be such region. Then, the grant event, g_i+, is inserted based on region $\langle r_i+, I_i \rangle$. The release event, r_i-, is inserted based on the minimum region having O_i as the input interface. Let β_i be the output interface of such region. Then, the completion event, g_i-, is inserted based on region $\langle r_i-, \beta_i \rangle$. Eventually, $\alpha_i = O_i$ and the four inserted events appear in sequence, creating a CSC conflict. In such cases, an extra signal is inserted to overcome this conflict. After inserting all channels, we must remove all states that are unreachable because of the exclusion behavior of the genex. Let's now formalize this procedure.

Procedure 4.15 (genex channel insertion)

1. *Determine $\langle \alpha_i, I_i \rangle$ as the minimum regions having I_i as the output interface. Insert r_i+ in G such that it appears after the α_i but before the I_i events.*
2. *Insert g_i+ in G such that it appears after the r_i+ event but before the I_i events.*
3. *Determine $\langle O_i, \beta_i \rangle$ as the minimum regions having O_i as the input interface. Insert r_i- in G such that it appears after the O_i but before the β_i events.*
4. *Insert g_i- such that it appears after the r_i- event but before the β_i events.*
5. *If $\alpha_i = O_i$ insert an extra signal to remove the CSC conflict caused by the channel insertion.*[5]

Procedure 4.16 (genex insertion)

1. *For $i = 1, 2, 3$, insert channel i of the genex as determined by procedure 4.15.*
2. *Remove unreachable states, due to genex behavior.*

Let's illustrate the use of a genex 3x2, controlling the non-persistencies in the example given by figure 4.23. The regions to be controlled are $R_1 = \langle \{b_1+, b_1-\} \rangle$, $R_2 = \langle \{b_2+, b_2-\} \rangle$ and $R_3 = \langle \{b_3+, a_3+\} \rangle$. As required by procedure 4.15, the values of α_i, and β_i, for $i = 1, 2, 3$, used to insert each channel are the following:

$$\alpha_1 = \{a_1+\}, \quad \beta_1 = \{a_1+\}$$
$$\alpha_2 = \{a_2+\}, \quad \beta_2 = \{a_2+\}$$
$$\alpha_3 = \{a_3+\}, \quad \beta_3 = \{r_3+\}$$

[5] For a detailed explanation of the transformation see chapter 5.

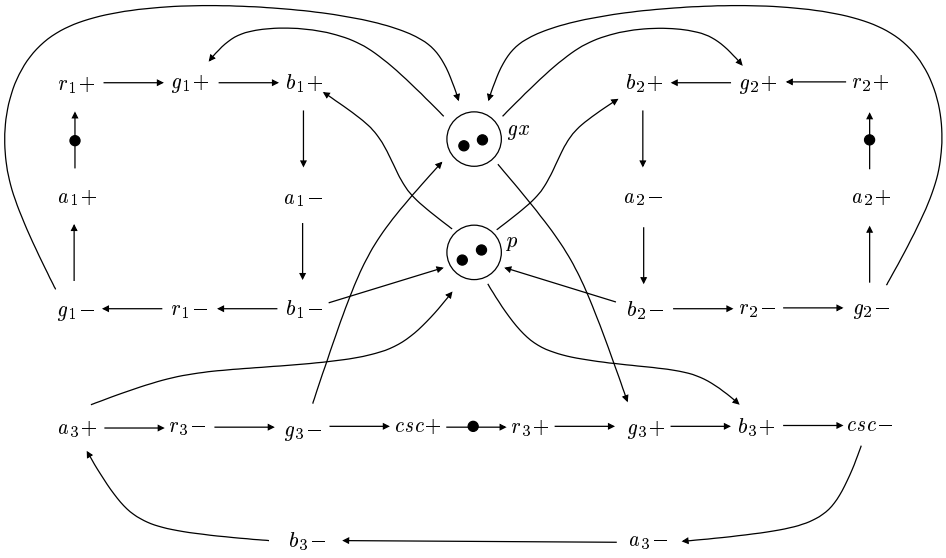


Figure 4.27: Transformed STG obtained from the one in figure 4.23 after insertion of a genex 3x2 to control access regions $R_1 = \langle\{b_1+, b_1-\}\rangle$, $R_2 = \langle\{b_2+, b_2-\}\rangle$. The corresponding state graph has 465 states, and so it is too big to make its graphical representation understandable.

Note that $\alpha_3 = O_3$, and so the minimum region having O_3 as the input interface appears to have r_3+, the request event, as the output interface. It is because of that that the CSC conflict appears. A signal, called csc, has been inserted to overcome the CSC conflict. Event $csc+$ was inserted based on region $\langle\{g_3\}-, \{r_3+\}\rangle$ and event $csc-$ was inserted based on region $\langle O_3, \{a_3-\}\rangle$, the minimum region having O_3 as the input interface. The transformed state graph have 465 states, too many to make its graphical representation understandable. Instead, an equivalent STG is depicted in figure 4.27.

The notion of exclusion of 2 out of 3 can be further generalized for the notion of *exclusion of k out of n*, with $k < n$: n concurrent regions are said to be in an exclusion relation of k out of n if every intersection of k different regions are different from the empty set, but any intersection of $k + 1$ different regions is the empty set. Formally,

Definition 4.17 (exclusion of k out of n)

Let G be a state graph and let $R_1, R_2, \cdots, R_n$ be n concurrent regions in G. Let $N_n = \{1, 2, \cdots, n\}$ be the set of natural numbers lower or equal n. Let $N_n^{(k)} \subset \wp(N_n)$ be the subset of the elements of the powerset of N_n with cardinality k. Regions $R_1, R_2, \cdots, R_n$ are said to be in an exclusion relation of k out of n *if the following two conditions hold:*

1. $\forall_{I \in N_n^{(k)}} : \left(\bigcap_{i \in I} R_i \right) \neq \emptyset$

2. $\forall_{I \in N_n^{(k+1)}} : \left(\bigcap_{i \in I} R_i \right) = \emptyset$

Definition 4.17 reduces to definition 4.13 if $n = 3$ and $k = 2$. It reduces to definition 4.5 if $n = 2$ and $k = 1$.

Also the notion of genex can be generalized to the notion of *genex* $n \times k$, for $k < n$. It is a device with n inputs and n outputs. It accepts requests on the input lines and its behavior is such that at most k grants are given at a time. The first k requests are automatically granted. The next must wait until a release-completion before it is granted by the device. This device can be used to control access to n concurrent regions, which are in an k out of n exclusive relation. The genex 2x1 corresponds to a 2-channel mutex.

4.4 Non-commutativities

Let $G = \langle S, E, \Theta, s_{in} \rangle$ be a state graph, where $E = V \cup \{+, -\}$, V being the set of signals. Let $a*, b* \in E$ be two signal transitions, associated with signals a and b, involved in a non-commutative relation in some state $s \in S$. Let s_{ab} and s_{ba}, with $s_{ab} \neq s_{ba}$ be the states reached from state s after the firing of traces $\langle a*, b* \rangle$ and $\langle b*, a* \rangle$ respectively. If the output events enabled in s_{ab} and s_{ba} are the same, the system (circuit) changes its outputs in exactly the same way in both states, and so from the synthesis point of view it is acceptable to implement them by the same state, that is, both states can be merged. Any differences in terms of enabled input events can also be correctly managed by the circuit, if the states are merged. Depending on the input event that has fired, the system evolves to the appropriate state. This merging of states is exactly what is already done by existing synthesis tools, and corresponds to the already mentioned CSC property. So, the analysis of non-commutativities must only be focused on cases where there are differences in terms of enabled output events in states s_{ab} and s_{ba}.

Figure 4.28 shows a state graph with such a conflict, where x_{15}, a_1+, and a_2+ play the roles of respectively s, $a*$, and $b*$. We will assume that b_1 and b_2 are output signals, thus introducing differences in terms of output events enabled in states s_{ab} and s_{ba}. Consider that the system is at state x_{15}. If

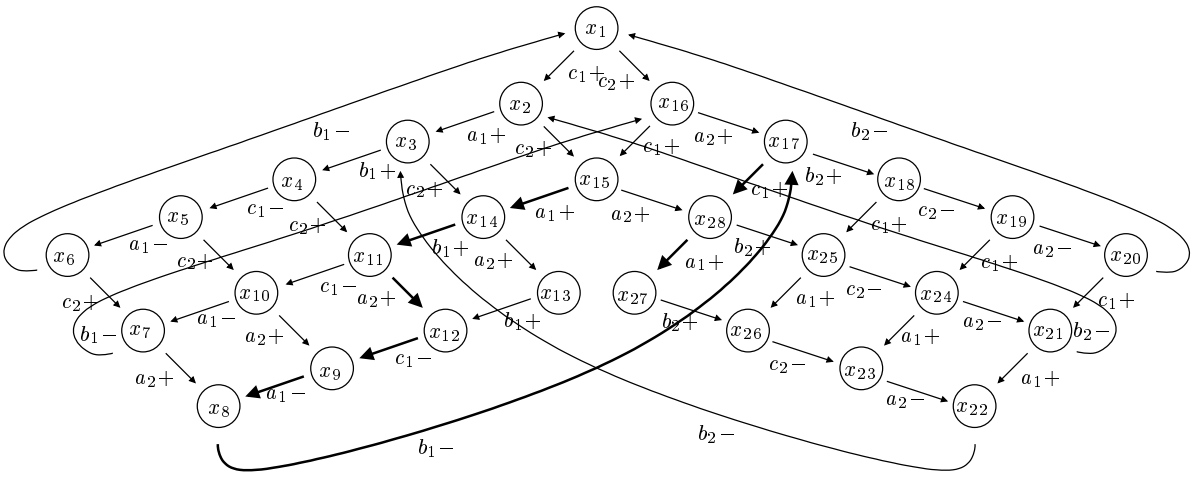


Figure 4.28: A state graph with a non-commutativity between events a_1+ and a_2+ in state x_{15}. Assuming b_1 and b_2 are output signals, this state graph manifest an SI-conflict.

a_1+ fires clearly before a_2+, the system evolves to the states on the left side, states x_{14} to x_7. On the contrary, if a_2+ fires clearly before a_1+ the system evolves to the states on the right side, states x_{28} to x_{21}. But, if a_1+ and a_2+ fire too close together, there is a "fight" between the two events. If the system senses first a_1+, it evolves to state x_{13}; otherwise it evolves to state x_{27}. In the former case event b_1+ is enabled, while in the latter b_2+ is enabled. Can this situation be simulated by a "fighting" between events b_1+ and b_2+, or to put it in other words, can we simulate this state graph by another having states x_{13} and x_{27} merged? Figure 4.29 shows this state graph with the two states merged. If a_1+ fires clearly before a_2, this system evolves to the left side, which is equivalent to the original behavior. The equivalence remains if a_2+ fires clearly before a_1+. If a_1+ and a_2+ fire too close together, this system evolves to state x_m where there is a choice between b_1+ and b_2+. After choosing a side to fall to, the behavior becomes equivalent to the original one. But, let's analyze how the two systems evolve due to the firing of trace $\langle a_1+, b_1+, a_2+, c_1-, a_1-, b_1-, c_1+, a_1+\rangle$, starting from state x_{15}. The original system evolves to state x_{27} and b_2+ fires next. However the merged system evolves to states x_m and either b_1+ and b_2+ can fire. But a_2+ has fired far before the last a_1+ and so this choice between b_1+ and b_2+ is unacceptable.

A different possible intervention to replace the non-commutativity is based on the notion of *obser-*

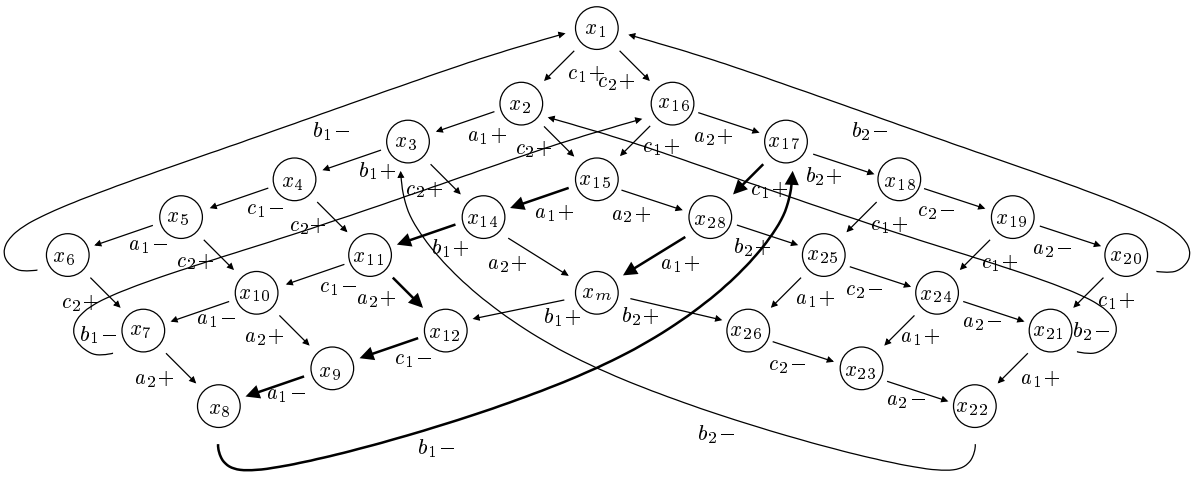


Figure 4.29: The state graph obtained from the one in figure 4.28 by the merging of states x_{13} and x_{27} into state x_m. The non-commutativity was transformed into a symmetric non-persistency between events b_1+ and b_2+.

vational equivalence [1]. This notion defines an equivalence between transition systems taking into account the existence of invisible transitions. Thus, the systems are only compared in terms of their "observable" behavior. In state graphs we can consider that transitions on internal signals are "invisible" from the environment point of view.

In [1] a formal definition of observational equivalence is given. It is based on a generalization of the notion of transition, to represent a transition from the point of view of the "observer". Let $G_1 = \langle S_1, E \cup N_1, \Theta_1, s_{1in} \rangle$ be a transition system, where E is a set of visible events and N_1 a set of invisible events. All events of N_1 are indistinguishable from the "observer" point of view, and so all of them can be represented by the single symbol ε. Thus, we can define the set $E^\varepsilon = E \cup \{\varepsilon\}$, where ϵ represents all invisible events. Based on E^ε, Arnold [1] defines *general transition* and *general set of transitions*.

Definition 4.18 (general transition)

Triple $\langle s, e, s' \rangle \in S_1 \times E^\varepsilon \times S_1$ *is a general transition if and only if*

- $e \in E$ *and there is a path from* s *to* s' *where one and only one of the events is* e*, while all the others belong to set* N_1*; or*

- $e = \varepsilon$ *and either there is a path from* s *to* s'*, with all events belonging to* N_1*, or* $s = s'$*.*

The set of all elements from $S_1 \times E^{\varepsilon} \times S_1$ satisfying definition 4.18 is called the *general set of transitions*, and is denoted by Θ_1^{ε}.

Based on the generalized set of transitions, Arnold defines observational equivalence. Let $G_2 = \langle S_2, E \cup N_2, \Theta_2, s_{2\,in} \rangle$ be another transition system, where E is the set of visible events and N_2 the set of invisible events. Define Θ_2^{ε} the same way we have define Θ_1^{ε}.

Definition 4.19 (observational equivalence)

Transition systems G_1 *and* G_2 *are* observationally equivalent *if there is a relation* $R \subseteq S_1 \times S_2$*, satisfying the following conditions [1]:*

1. *for every state* $s_1 \in S_1$*, there exists* $s_2 \in S_2$ *such that* $\langle s_1, s_2 \rangle \in R$*;*
2. *for every state* $s_2 \in S_2$*, there exists* $s_1 \in S_1$ *such that* $\langle s_1, s_2 \rangle \in R$*;*
3. *for every pair* $\langle s_1, s_2 \rangle \in R$*, for every general transition* $\langle s_1, e, s_1' \rangle \in \Theta_1^{\varepsilon}$*, there exists* $s_2' \in S_2$ *such that* $\langle s_1', s_2' \rangle \in R$ *and* $\langle s_2, e, s_2' \rangle \in \Theta_2^{\varepsilon}$*; and*
4. *for every pair* $\langle s_1, s_2 \rangle \in R$*, for every general transition* $\langle s_2, e, s_2' \rangle \in \Theta_2^{\varepsilon}$*, there exists* $s_1' \in S_1$ *such that* $\langle s_1', s_2' \rangle \in R$ *and* $\langle s_1, e, s_1' \rangle \in \Theta_1^{\varepsilon}$*.*

Observational equivalence and signal projection are connected by the following theorem.

Theorem 4.20

Let $G = \langle S, E, \Theta, s_{in} \rangle$ *be a state graph, with* $E = V \times \{+, -\}$*, being* V *the set of signals. Let* $V' \subseteq V$ *be a subset of signals. Assuming* $(V - V') \times \{+, -\}$ *are invisible events,* G *and* $G' = G \Downarrow V'$ *are observational equivalents.*

Proof

The proof will be done constructing a relation R *satisfying the 4 conditions of definition 4.19. By the definition of signal projection (definition 3.5), set* S *is partitioned into a set of equivalent classes. Let* $[s] \in S'$ *be the equivalent class of state* $s \in S$*. Then construct* R *as follows:*

$$R = \{\langle s, s' \rangle \ : \ s \in S \ \wedge \ s' = [s]\}$$

Clearly, conditions 1 and 2 of definition 4.19 are satisfied.

Let now evaluate satisfying of condition 3. Assume $\langle x, [x]\rangle \in R$ *and* $t = \langle x, e, y\rangle \in \Theta^{\varepsilon}$. *If* $e \in E' = V' \times \{+, -\}$, t *represents a path in* G, *where one and only one event belongs to* E' *while all the others belong to* $E - E'$. *Because any two states connected by an event from* $E - E'$ *belong to the same equivalent class, the image of* t *by the projection is transition* $\langle [x], e, [y]\rangle$. *But* $\langle y, [y]\rangle \in R$ *and* $\langle [x], e, [y]\rangle \in (\Theta')^{\varepsilon}$, *and thus condition 3 is satisfied. If* $e = \varepsilon$, t *represents a path in* G, *where all events are from* $E - E'$. *Its image by the projection is state* $[x]$, *that is,* $[y] = [x]$. *Because* $\langle [x], \varepsilon, [x]\rangle \in (\Theta')^{\varepsilon}$, *condition 3 is also satisfied. If* $e = \varepsilon$ *and* $x = y$, t *represents a state in* G. *Its image by the projection is state* $[x]$. *Again, Because* $\langle [x], \varepsilon, [x]\rangle \in (\Theta')^{\varepsilon}$, *condition 3 is satisfied.*

Finally, let evaluate the satisfying of condition 4. Assume $\langle x, [x]\rangle \in R$ *and* $t = \langle [x], e, [y]\rangle \in (\Theta')^{\varepsilon}$. *In* G' *there is no invisible events. Thus there are only two cases for* t*: a transition* $\langle [x], e, [y]\rangle \in \Theta'$ *or a transition* $\langle [x], \varepsilon, [x]\rangle$. *In the first case, we have* $\langle y, [y]\rangle \in R$ *and* $\langle x, e, y\rangle \in \Theta$, *and thus condition 4 is satisfied. In the second case, we have* $\langle x, \varepsilon, x\rangle \in \Theta^{\varepsilon}$ *and again condition 4 is satisfied.*

Since relation R *satisfy all conditions of definition 4.19, state graphs* G *and* G' *are observationally equivalents.*

The circuit in figure 4.30 was constructed from the one in figure 4.28 by adding two new internal signals, n_1 and n_2, in order to control the cause of non-commutativity. The projection of this state graph by the non-internal signals is isomorphic to the original specification (figure 4.28), making them *observationally equivalent*.

Let's evaluate how the original trace $\langle a_1+, b_1+, a_2+, c_1-, a_1-, b_1-, c_1+, a_1+\rangle$ behaves in it. It is transformed into trace $\langle a_1+, n_1+, b_1+, a_2+, c_1-, a_1-, n_1-, b_1-, c_1+, n_2+, a_1+\rangle$. Starting at state x_{20}, this trace makes the system evolve to state x_{36}, where only event b_2+ is enabled. This is equivalent to the behavior of the original specification. But, how behaves this system if a_1+ and a_2+ fire too close together? According to this specification this cannot happen, because between the firing of a_1+ and a_2+ there must exist time enough to fire either n_1+ or n_2+. But according to the original specification it can happen. Thus, specifications in figures 4.28 and 4.30 are equivalent if we can assume that the two inputs events never come simultaneously,

Another different approach to implement the behavior in figure 4.28 is based on the usage of a new

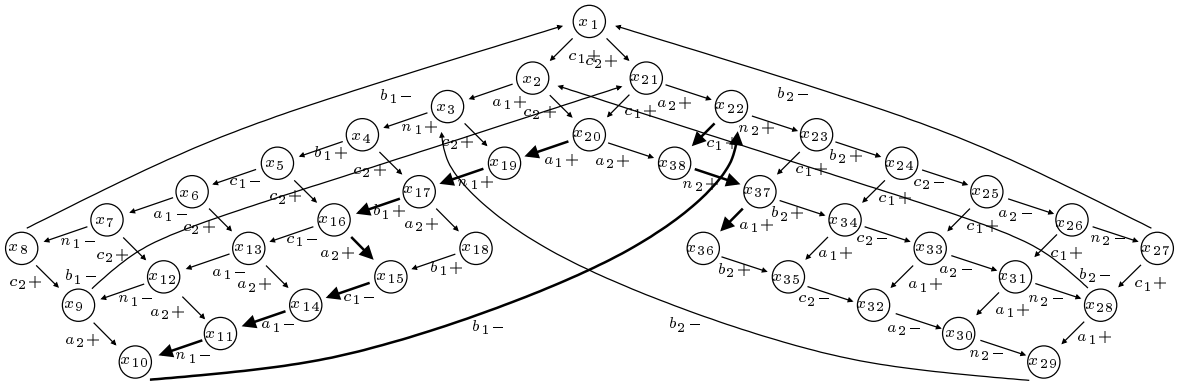
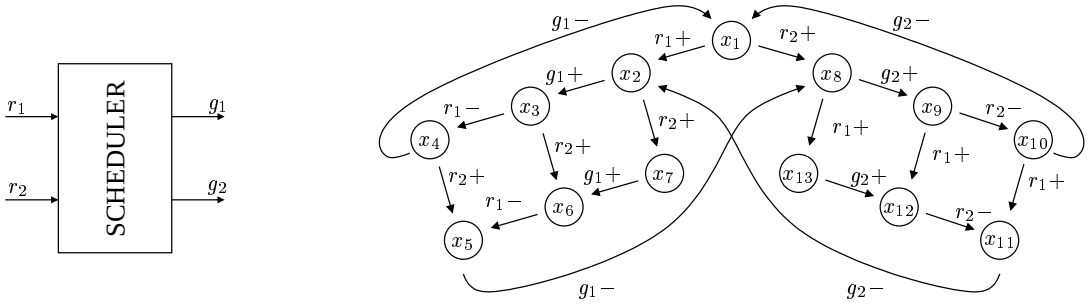


Figure 4.30: A state graph observationally equivalent to the one in figure 4.28, without any non-commutativity.

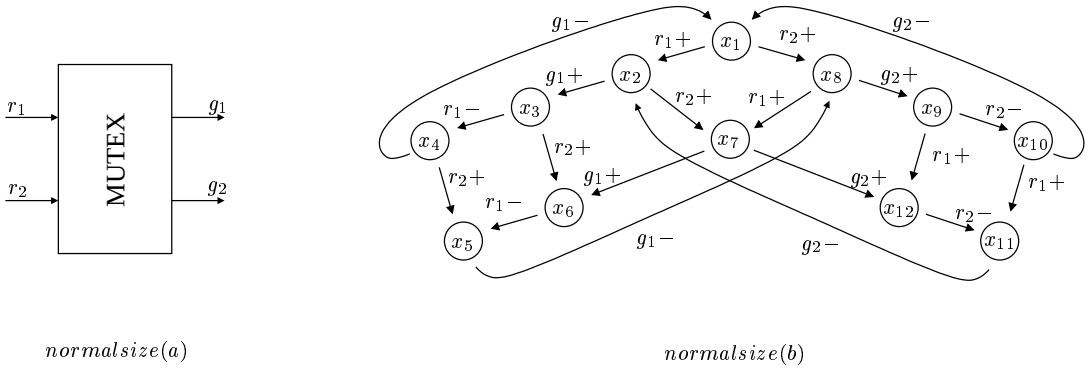
special arbitration device, called a *scheduler*. It is based on the two-agent scheduler, described in [2], and is used by a resource owner to manage its availability between two agents. It is a two-input, two-output device (figure 4.31.a), which accepts requests at any time and guarantees grants under mutual exclusion. For a four-phase protocol its behavior is as follows. Positive transitions on the input lines represent requests to the resource owner. Positive transitions on the output lines represent usage grants. Negative transitions on the input and output lines represent respectively resource release and cycle completion. The state graph description of this device is depicted in figure 4.31.b. We will call this device a *two-channel scheduler*, or simply a *scheduler*. Note that the behavior of a two-channel scheduler has an intrinsic non-commutativity, which is assumed to be correctly managed by the device itself.

The behavior of a two-channel scheduler is quite similar to the behavior of a two-channel mutex. They both guarantee mutual exclusion of their grants. The difference lies on the fact that the scheduler has memory, as can be understood by the following trace. Consider the application of trace $\langle r_1+, g_1+, r_2+, r_1-, g_1-, r_1+\rangle$ to the state graph of a two-channel scheduler (figure 4.31.b) and to the state graph of a two-channel mutex (figure 4.32.b), starting at state x_1 in both cases. The scheduler evolves to state x_{13}, where only the grant of channel two is enabled. The mutex evolves to state x_7, where both grants are enabled. However, it is quite improbable to the mutex to evolve to state x_7 in such situations. Still analyzing the same trace, when event g_1- occurs and the mutex evolves to state x_8, input r_1 is at 0 and input r_2 is at 1. The mutex then decides to raise output g_2. Even if the



(a) (b)

Figure 4.31: A two-channel scheduler: (a) block diagram; (b) state graph description.



$normalsize(a)$ $normalsize(b)$

Figure 4.32: A two-channel mutex: (a) block diagram; (b) state graph description.

input r_1 raises now, the mutex should not change its decision. Thus, in many practical situations, a two-channel scheduler can be implemented by a two-channel mutex.

Let us conclude our analysis on non-commutativities considering the different combinations for the two events: input-input, output-output and input-output. If a and b are both input signals, then events $a*$ and $b*$ are issued by the environment. Thus we cannot presume any temporal interrelation between them. Somehow the non-commutativity must be controlled by a scheduler. The same occurs if one of the signals, say a, is an input and the other, say b, an output.

Consider now that both a and b are output signals. When the system is in state s events $a*$ and $b*$ are enabled and, since both are output signal transitions, they both will fire. More, they for sure will fire close together in time. So, it seems quite acceptable that both states s_{ab} and s_{ba} could be merged.

Let's illustrate this with an example. Take the state graph depicted in figure 4.28 and let a_1, a_2, a_3 and a_4 be output signals and c_1 and c_2 input signals. There is a non-commutativity between output events, namely between a_1+ and a_2+. State x_{13} and x_{27} have different output events enabled, thus there is an SI-conflict associated with the non-commutativity. Assume that after some firing sequence of events, state x_{28} is reached. Output events b_2+ and a_1+ are simultaneously enabled. It is quite obvious to accept that these two events are going to occur next and simultaneously. Thus if we merge states x_{13} and x_{27} (obtaining the state graph in figure 4.29), it is quite improbable that event b_1+ will ever fire.

State graphs in figures 4.28 and 4.29 can be implemented by the circuits depicted in figure 4.33.a and 4.33.b, respectively. Consider that both circuits are in a state such that signals a_1, b_1 and c_1 are at 1 and signals a_2, b_2 and c_2 are at 0. This is a stable state and circuits will remain in it until one of the input signals changes value. This corresponds to state x_1 in the state graphs of figures 4.28 and 4.29. Now apply in both cases the following firing sequence of events $\langle c_2+, a_2+, c_1-, a_1-, b_1-\rangle$. Both systems evolve to state x_{17}. Consider now that c_1+ fires. In the circuit on the left side, the scheduler guarantees that b_2+ will fire before b_1+, because c_2+ clearly occurred before c_1+. But in the circuit with the mutex, the same must happen. By that time c_1+ fires, b_2+ is already firing, and so b_1+ will have to wait.

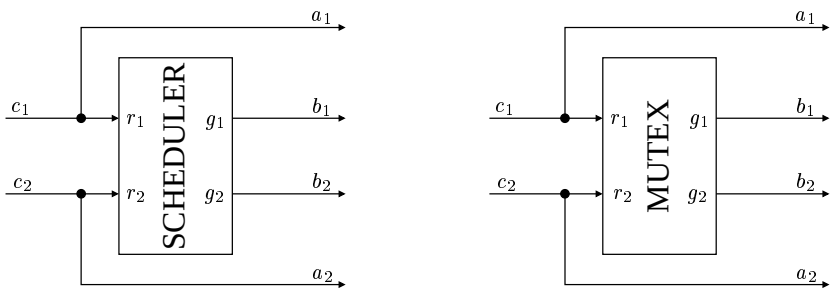


Figure 4.33: Circuit implementations for the state graphs of figures 4.28 (scheduler) and 4.29 (mutex) assuming that c_1 and c_2 are input signals and that a_1, a_2, b_1 and b_2 are output signals.

4.5 Conclusions

State graphs can have SI-conflicts of two types: non-persistencies and non-commutativities. Non-persistencies can be controlled by means of the insertion of mutual exclusion (mutex) devices or general exclusion (genex) devices. The former is a well known device, largely referred in the literature, which controls access under mutual exclusion to a shared resource. The latter defines a more general class of exclusion among different agents. A genex $n \times k$, for $n < k$, controls the access to k resources shared by n different agents. The insertion of genexes in a state graph with non-persistencies is based on region analysis. If there are n concurrent regions in a k out of n exclusive relation, then a genex $n \times k$ can be used to control access to these regions.

The study of non-commutativities is still not concluded. By now, we can say that the scheduler appears as a good candidate to control state graphs with non-commutativities. We have not developed a real circuit with exactly the behavior of a scheduler. However, we have shown that in many practical situations a scheduler can be implemented by a mutex.

The use of schedulers can also enlarge the class of specifications accepted by *petrify*. For example, the specification in figure 4.2 has an irreducible CSC conflict, and so is not solved by *petrify*. But the only differences between this state graph and the state graph of the two-channel scheduler (figure 4.31.b) are two input signal transitions, missing in the former. Thus, the behavior in figure 4.2 can be simply implemented by a two-channel scheduler.

Chapter 5

Synthesis of Non-Persistent Specifications

In chapter 4 we have analyzed state graph circuit specifications with two types of SI-conflicts: non-persistence and non-commutativity. For the former we went deeper in the study and we proposed a methodology to control the conflict points of a specification by means of insertions of mutexes or genexes. We call *conflict point* a maximal set of concurrent regions in an exclusion relation of k out of n, where n is the number of concurrent regions and $k < n$. In this chapter, we continue that work in order to develop algorithms suitable to be used in an automatic synthesis tool.

Tools for the automatic synthesis of speed-independent asynchronous circuits already exist. However, they impose the specification to be free of SI-conflicts. Examples of these tools are **SIS** ([73]) and *petrify* ([22, 20]), *petrify* been the most recent one and actually covering a wider class of specifications. In order to cover specifications with SI-conflicts, we do not intend to develop a new tool. Actually, it makes sense to benefit from the power of existing synthesis tools. Thus, we intend to adapt *petrify*, extending the class of specifications it accepts at its entry point such that SI-conflict specifications are covered.

Petrify accepts a circuit specification either as a signal transition graph (STG) description or a state graph description. Our methodology is developed in the state graph domain. Thus, if a specification is given as an STG description, we need to translate it to a state graph description before our procedures can be applied. This is not a problem, for *petrify* can be used to make the translation.

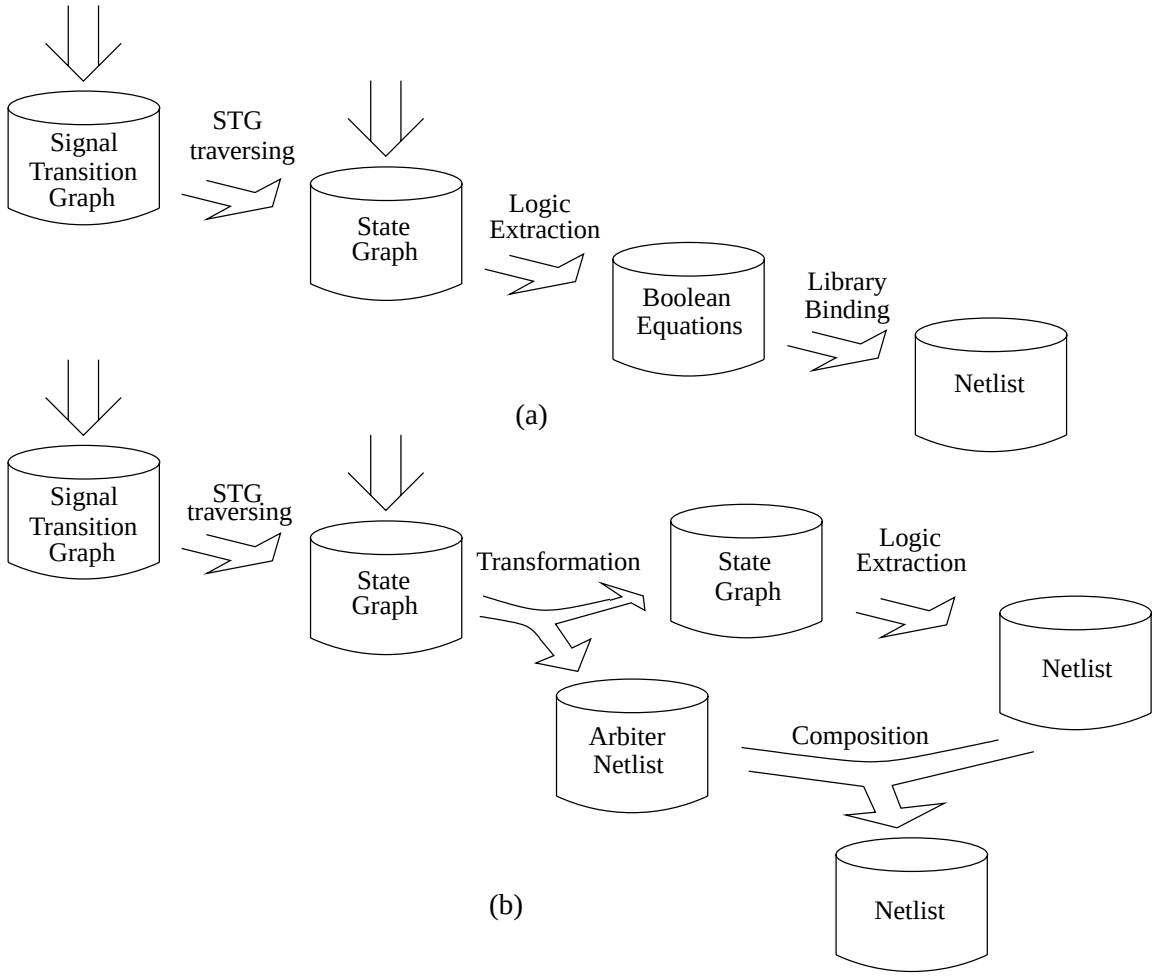


Figure 5.1: (a) Data flow of a typical synthesis tool for speed-independent asynchronous circuits, where the specification must be absent of SI conflicts. (b) Data flow of proposed synthesis approach, where the specification can contain SI conflicts of the non-persistent type.

5.1 Synthesis Overview

The data flow diagram of *petrify* in what is concerned with the synthesis of speed independent asynchronous circuits is given in figure 5.1.a. It is basically composed of three processing stages: *STG traversing*, *logic extraction* and *library binding*. At the entry point *petrify* accepts a signal transition graph specification of the circuit, in the *ASTG* file format, a file format borrowed from SIS ([73]). The STG is traversed and a state graph specification of the circuit is obtained. Eventually the initial specification can be given as a state graph, using a proper specification file format introduced by *petrify*, in

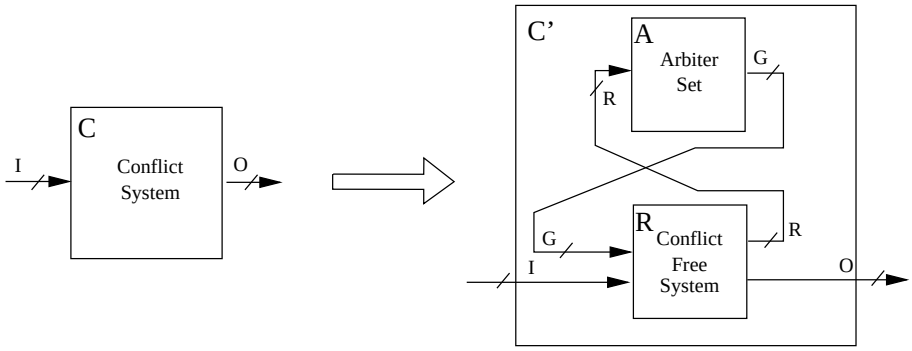


Figure 5.2: Overview of synthesis transformation process.

which case the traversing phase is skipped. If the state graph specification holds speed-independence *petrify* generates the set of boolean equations for the output and internal signals of the circuit[1]. Finally these boolean equations can be mapped onto a given library of components. On section 1.6.5 we have presented a wider overview of the speed-independent synthesis phases.

The overall synthesis flow of our proposed approach to cover specifications with SI-conflicts is depicted in figure 5.1.b. Two new processing stages appear in this flow diagram relatively to the flow diagram of petrify: *transformation* and *composition*.

The *transformation* process takes a state graph specification as its input data. If there is no SI-conflict it delivers the same state graph as its output data. If there are SI-conflicts it delivers a set of arbiters and a state graph description as its output data. The arbiters must manage all SI-conflicts of the original specification. The state graph description must be free from SI-conflicts, so it can be submitted to existing synthesis tools. Figure 5.2 graphically represents the purpose of the transformation process. If $\mathcal{C}$ is a system specification with SI-conflicts, it must be transformed to some $\mathcal{C}'$, composed of two sub-systems $\mathcal{A}$ and $\mathcal{R}$ such that:

1. $\mathcal{A}$ is a set of arbiter devices which encapsulate all SI-conflicts;
2. $\mathcal{R}$ is a conflict-free state graph, suitable to be synthesized using *petrify*.
3. the composite behavior of $\mathcal{R}$ and $\mathcal{A}$ is equivalent to the behavior of $\mathcal{C}$;

[1] If committed to do so, *petrify* can also generate the set of boolean equations if the speed-independence property is not held by the specification. In such cases the produced implementation only works correctly under some time constraints.

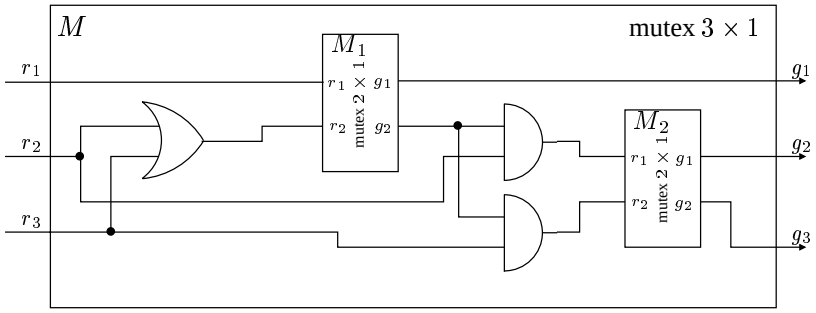


Figure 5.3: A genex (mutex) 3×1, built up of 2 genexes (mutexes) 2×1.

The arbiters are assumed to be library components. All that the *composition* process has to do is to connect some output of $\mathcal{R}$ to the inputs of the arbiters and the outputs of the arbiters to some inputs of $\mathcal{R}$. This is a quite trivial task. The rest of the chapter is thus concentrated on the transformation process.

5.2 Arbiters

The arbiter plays a central role in the transformation process. For each identified SI-conflict we must select an appropriated arbiter. In sequel we assume our library contains the general exclusion device, genex $n \times k$, for every n and every k, with $k < n$.

The genex (mutex) 2×1 is a quite well known device [72], we have already presented in chapter 1 (see section 1.7). Its CMOS circuit implementation is depicted in figure 1.22. With genexes 2×1 we can build the genex $n \times 1$, for $n > 2$. Figure 5.3 shows how to build a genex 3×1, using two genexes 2×1. Channel 1 of mutex M_1 directly implements channel 1 of the genex (mutex) 3×1. Channel 2 of mutex M_1 and mutex M_2 together implement channels 2 and 3 of the genex 3×1. If a request is issued in line r_1 and it is granted by mutex M_1, output g_2 of mutex M_1 is kept low and thus, because of the AND gates, no requests from lines r_2 or r_3 can reach mutex M_2. If, in result to a request in lines r_2 or r_3, output g_2 of M_1 goes high, mutex M_2 guarantees only one of these requests is granted.

We can easily extend the structure depicted in figure 5.3 in order to built a mutex 4×1. One way of achieving that is mounting around channel 1 of mutex M_1 a structure equal to the one mounted

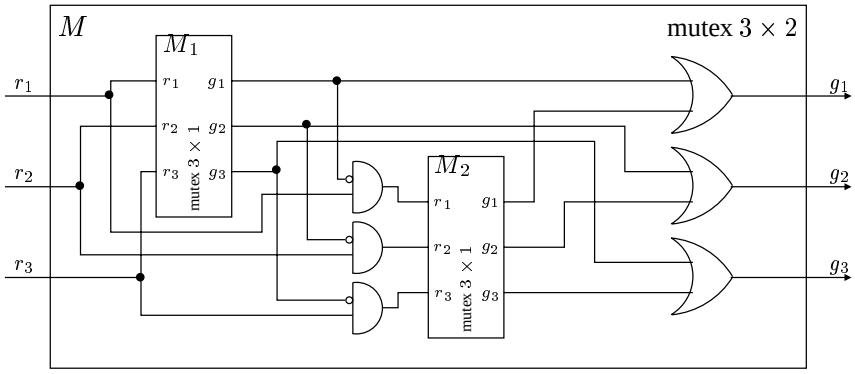


Figure 5.4: A genex 3×2, built up of 2 genexes (mutexes) 3×1.

around channel 2. A symmetric structure is obtained in this way. Alternatively, we can use a mutex 3×1 in place of the mutex M_1. Channels 1 and 2 of the new mutex M_1 directly implement channels 1 and 2 of the mutex 4×1. Channel 3, in conjunction with mutex M_2, the AND gates and the OR gate, implements channels 3 and 4 of the mutex 4×1. Using the last approach we can inductively define the mutex $n \times 1$, for any $n > 2$: a mutex $n \times 1$ can be built from a mutex $(n-1) \times 1$, a mutex 2×1, two AND gates and an OR gate. Let M represent the mutex $n \times 1$, M_1 the mutex $(n-1) \times 1$ and M_2 the mutex 2×1. The following expressions show how to build M from M_1 and M_2, where we use $C.x$ to denote signal x of component C:

$$
\begin{aligned}
&M_1.r_i = M.r_i, \quad \text{for } i = 1, 2, \cdots, n-2 \\
&M.g_i = M_1.g_i, \quad \text{for } i = 1, 2, \cdots, n-2 \\
&M_1.r_{n-1} = M.r_{n-1} \ \vee \ M.r_n \\
&M_2.r_1 = M_1.g_{n-1} \ \wedge \ M.r_{n-1} \\
&M_2.r_2 = M_1.g_{n-1} \ \wedge \ M.r_n \\
&M.g_{n-1} = M_2.g_1 \\
&M.g_n = M_2.g_2
\end{aligned}
$$

Using two genexes 3×1 we have a proposal to build a genex 3×2. It is depicted in figure 5.4. The circuit for each one of the three channels is the same. Input signal r_i applies directly to input r_i of mutex M_1 and applies to input r_i of mutex M_2 through an AND gate. The other input of this AND gate is controlled by output g_i of mutex M_1. A grant to request r_i is given whenever a grant is given

by one of the mutexes.

Consider that initially all inputs and outputs are low. Now let r_1 raise. Then input r_1 of M_1 raises and input r_1 of M_2 also raises after the delay of the AND gate. Output g_1 of M_1 goes to high, and so does output g_1 of the genex 3×2. Eventually, output g_1 of M_2 can go high for a short period of time. This is associated to the period of time while input r_1 is at high but output g_1 of M_1 is at low. Any other request that comes next is routed to mutex M_2. Because input r_1 of M_2 is at low, one, but only one, request will be granted. Thus we can have at most two requests granted.

Consider now that two requests, say r_1 and r_2, arrive simultaneously. Mutex M_1 will grant one of them. Mutex M_2 will necessarily grant the other, even if initially it has decided to grant the same channel as mutex M_1. That's because the grant from mutex M_1 removes the corresponding request from mutex M_2. If the third request comes next, its grant must wait until a mutex is available.

Finally consider that all three requests come at the same time. Similarly to the previous explanation, mutex M_1 will grant one of the requests, mutex M_2 will grant another, while the third has to wait.

In the same way we have built a genex 3×2 from two mutexes 3×1, we can build a genex $n \times 2$ from two mutexes $n \times 1$, for $n > 3$. Let M represent the genex $n \times 2$ and M_1 and M_2 two genexes $n \times 1$. The following expressions show how to build M from M_1 and M_2:

$$
\begin{aligned}
&M_1.r_i = M.r_i, \quad \text{for } i = 1, 2, \cdots, n \\
&M_2.r_i = M.r_i \ \wedge \ \overline{M_1.g_i}, \quad \text{for } i = 1, 2, \cdots, n \\
&M.g_i = M_1.g_i \ \vee \ M_2.g_i, \quad \text{for } i = 1, 2, \cdots, n.
\end{aligned}
$$

A genex $n \times 3$ can be built using the same construction as for the genex $n \times 2$ but with one of the genexes, M_1 or M_2, replaced with a genex $n \times 2$. Then, inductively, we can build a genex $n \times k$ from one genex $n \times (k-1)$ and one genex $n \times 2$. For instance, the equations given for the genex $n \times 2$ can be used to represent a genex $n \times k$, if M_1 represents a genex $n \times (k-1)$ instead of a genex $n \times 1$.

5.3 Transformation Process

The approach used to accomplish the transformation process is to manage one conflict point at a time. Given a conflict specification, a conflict point is selected and the specification is transformed in

order to transfer the conflict to an appropriate arbiter. After this step we should have an arbiter plus a transformed specification. If this transformed specification still has conflicts, the process is repeated with one of the remaining conflicts. The process continues until all conflicts are transferred to the appropriate arbiters. This is formalized by the following procedure:

Procedure 5.1 (transformation process)

1. *Let G be the original state graph specification.*
2. *While G has conflicts*
3. *Select a conflict point and select a genex to manage it.*
4. *Transform G into G' in order the conflict get managed by the genex.*
5. *Let $G = G'$.*
6. *Return G.*

Now we have to concentrate on step 3 of procedure 5.1. A conflict point is characterized by the existence of n concurrent regions in an exclusion relation of k out of n (see section 4.2). Eventually, one of the regions involved in the conflict point is the empty region. This conflict point can be controlled by a genex $n \times k$.

If $k < n - 1$, the existence of a conflict point of order k out of n implies the existence of n conflict points of order k out of $n - 1$. This is stated by the following theorem.

Theorem 5.2

Let G be a state graph and let $R_1, R_2, \cdots, R_n$ be n concurrent regions in G. If regions $R_1, R_2, \cdots, R_n$ are in an exclusion relation of k out of n and $k < n - 1$, then any subset of $n - 1$ of these regions is in an exclusion relation of k out of $n - 1$.

Proof

Let $N_n = \{1, 2, \cdots, n\}$ be the set of natural numbers lower than or equal to n, let $j \in N_n$ and let $A = N_n - \{j\}$. Let $N_n^{(k)} \subset \wp(N_n)$ be the subset of the elements of the power-set of N_n with cardinality equal k. Since $A \subset N_n$, we have $A^{(k)} \subset N_n^{(k)}$. Then from equation 1 of definition 4.17, we have

$$\forall_{I \in A^{(k)}} : \left(\bigcap_{i \in I} R_i \right) \neq \emptyset.$$

Similarly, $A^{(k+1)} \subset N_n^{(k+1)}$ and, from equation 2 of definition 4.17, we have

$$\forall_{I \in A^{(k+1)}} : \left(\bigcap_{i \in I} R_i \right) = \emptyset.$$

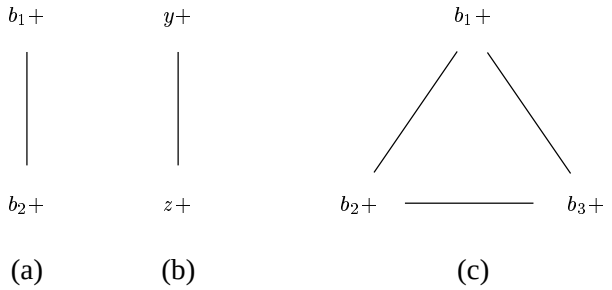


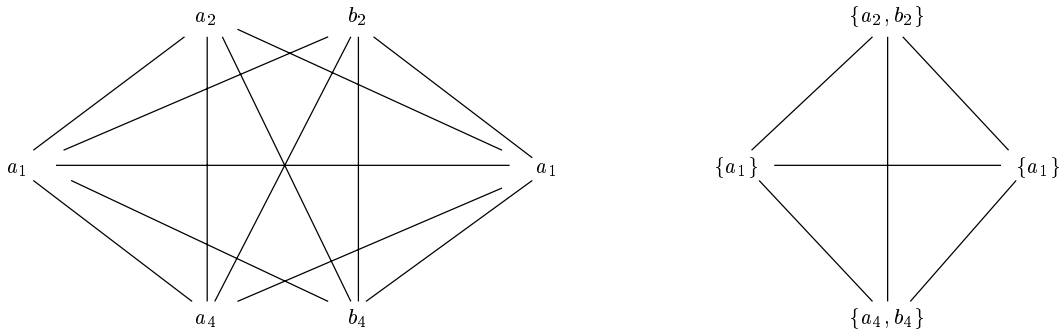
Figure 5.5: Non-persistence graphs for some examples from chapter 4, namely the state graphs from: (a) figures 4.17 and 4.21; (b) figure 4.19; and (c) figure 4.23.

Thus, the subset of $n-1$ regions defined by A is in an exclusion relation of k out of $n-1$. Since j is any element from N_n, the proof is concluded.

The previous theorem shows that conflict points must be searched for from higher to lower orders. This search can be done based on graph analysis. If two concurrent regions are in an exclusion relation of 1 out of 2, then events of the input interfaces of both regions are involved in some non-persistence relation. If three concurrent regions are in an exclusion relation of k out of 3, with $k < 3$, then events from the input interface of any one of the 3 regions are involved in some non-persistence relation with events from the input interface of any of the other regions. Generalizing, there are non-persistence relations between events from the input interfaces of any two of n concurrent regions in an exclusion relation of k out of n, for $k < n$. This suggest the construction of a *non-persistence graph*, where vertices represent events and edges connect events involved in some non-persistence relation. In figure 5.5 the non-persistence graphs for 4 state graphs used as examples in section 4.2 are drawn.

Exclusions relations in a state graph G are responsible for complete subgraphs in the G's non-persistence graph.[2] Looking in the opposite direction, complete subgraphs in the non-persistence graph can be associated with exclusion relations in G. However, we have to look for complete subgraphs of maximal order. A complete subgraph of order n contains n complete subgraphs of order $n-1$, $n*(n-1)/2$ complete subgraphs of order $n-2$, and so on. Then an exclusion relation of $(n-1)$ out of n generates complete subgraphs of orders from 2 to n. Only the one of order n represents the exclusion relation.

[2] A subgraph is *complete* if any two different vertices are adjacent.



(a) (b)

Figure 5.6: Non-persistence graphs for a state graph with 4 concurrent regions, involved in an exclusion relation of 2 out of 4, the input interfaces of the 4 regions being $I_1 = \{a_1\}$, $I_2 = \{a_2, b_2\}$, $I_3 = \{a_3\}$ and $I_4 = \{a_4, b_4\}$: (a) Obtained directly from non-persistence analysis of the original state graph. (b) Obtained from the previous after merging vertices not self-adjacent, but adjacent to exactly the same vertices.

Another difficulty arises if one or more of the concurrent regions have non-singleton input interfaces. Assume, for instance, we have a state graph G with 4 concurrent regions, $R_i = \langle I_i, O_i \rangle$, for $i = 1, 2, 3, 4$, involved in an exclusion relation of 2 out of 4 and let $I_1 = \{a_1\}$, $I_2 = \{a_2, b_2\}$, $I_3 = \{a_3\}$ and $I_4 = \{a_4, b_4\}$. The non-persistence graph for G is depicted in figure 5.6.a. In this graph we can identify 4 complete subgraphs of order 4, corresponding to the following sets of vertices: $\{a_1, a_2, a_3, a_4\}$, $\{a_1, b_2, a_3, a_4\}$, $\{a_1, a_2, a_3, b_4\}$ and $\{a_1, b_2, a_3, b_4\}$. We can also identify that vertices a_2 and b_2 are not connected to each other, but both are connected to exactly the same vertices. The same is true for vertices a_4 and b_4. If we merge vertices a_2 with b_2 and a_4 with b_4, we obtain the non-persistence graph in figure 5.6.b. This one exactly represents the exclusion relation in G.

The conflict point searching procedure must start by drawing the non-persistence graph. Next, it must merge equivalent vertices. Two vertices are *equivalent* if they are not adjacent one to each other and they are adjacent to exactly the same set of vertices. Third, it must identify a maximal complete subgraph. This complete subgraph should represent a conflict point, that is, it should represent a set of n concurrent regions involved in an exclusion relation of k out of n.

At this moment we must determine the values of n, k and the regions interfaces. The value of n corresponds to the order of the complete subgraph. The input interfaces are given by the vertices of

the complete subgraph. But what about the values of k and the output interfaces? We do not have an analytical way to determine them. What we propose is to explore regions in the state graph, starting with the known input interfaces, in order to determine a value for k and maximal regions which satisfy the two equations of definition 4.17.

More than one solution can be found after the exploration. This will be illustrated with an example. Consider the system depicted in figure 5.7. It is a state graph G with 6 signals — 3 inputs and 3 outputs —, 54 states and 145 transitions. These dimensions make it impossible to represent the state graph picture in a legible way. Thus, we have decided to give its description both as a Petri net and as an ASTG description, a textual format accepted by *petrify*. Consider the Petri net description. Places p_1 and p_2 are 2-bounded. If we fire events a_1+, a_2+ and a_3+, the net evolves to a marking where places p_1 and p_2 have two tokens each and events z_1+, z_2+ and z_3+ are enabled to fire. If any one of these events fires, one token is removed from both p_1 and p_2 and the net moves to a marking where the other two events are in a symmetrical non-persistence. Actually, the state graph description contains 8 symmetrical non-persistences, 3 between events z_1+ and z_2+, at states x_{51}, x_{52} and x_{53}, 2 between z_1+ and z_3+, at states x_{46} and x_{50}, and 3 between z_2+ and z_3+, at states x_{14}, x_{28} and x_{38}.

The non-persistence graph of G is given in figure 5.8.a. It is a complete graph of order 3, and so we are looking for 3 concurrent maximal regions, $R_i = \langle I_i, O_i \rangle$, for $i = 1, 2, 3$, involved in an exclusion relation of k out of 3, with $k < 3$. Exploring the region space of G, starting with $I_1 = \{z_1+\}$, $I_2 = \{z_2+\}$ and $I_3 = \{z_3+\}$, we have got $k = 2$ and two solutions for O_1, O_2 and O_3. They are:

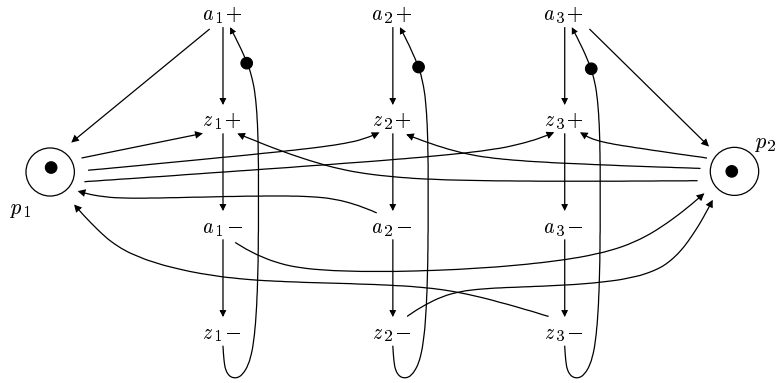
$$O_1 = \{a_1+\}, \quad O_2 = \{a_2-\}, \quad O_3 = \{z_3-\}$$

and

$$O'_1 = \{a_1-\}, \quad O'_2 = \{z_2-\}, \quad O'_3 = \{a_3+\}$$

We have looked for maximal regions, so these two solutions actually represent different conflict points. Each must be controlled by a different genex. If only one conflict point is controlled, some non-persistences are not covered.

This fact can be easily observed adding some new information to the non-persistence graph. Let extend the definition of non-persistence graph by adding a labeling function to the edges. This function maps every edge into the subset of states, where the non-persistence represented by the edge




```
#
# State graph with 54 states,
# 145 transitions and
# 8 symmetrical non-persistencies:
# 3 between z1+ and z2+
# in states x51, x52 and x53
# 2 between z1+ and z3+
# in states x46 and x50
# 3 between z2+ and z3+
# in states x14, x28 and x38
#
.model tt-3x2
.inputs a1 a2 a3
.outputs z1 z2 z3
.state graph
x01 a1+ x39 z1+ x29 a1- x15 z1- x01
x02 a1+ x40 z1+ x30 a1- x16 z1- x02
x03 a1+ x41 z1+ x31 a1- x17 z1- x03
x04 a1+ x42 z1+ x32 a1- x18 z1- x04
x05 a1+ x43
x19 z1- x05
x06 a1+ x44
x20 z1- x06
x07 a1+ x45
x21 z1- x07
x08 a1+ x46 z1+ x33 a1- x22 z1- x08
x09 a1+ x47
x23 z1- x09
x10 a1+ x50 z1+ x34 a1- x24 z1- x10
x11 a1+ x51 z1+ x35 a1- x25 z1- x11
x12 a1+ x52 z1+ x36 a1- x26 z1- x12
x13 a1+ x53 z1+ x37 a1- x27 z1- x13
x14 a1+ x54 z1+ x38 a1- x28 z1- x14
x01 a2+ x11 z2+ x09 a2- x05 z2- x01
x02 a2+ x12
x06 z2- x02
x03 a2+ x13
x07 z2- x03
x04 a2+ x14 z2+ x10 a2- x08 z2- x04
x15 a2+ x25 z2+ x23 a2- x19 z2- x15
x16 a2+ x26
x20 z2- x16
x17 a2+ x27
x21 z2- x17
x18 a2+ x28 z2+ x24 a2- x22 z2- x18
x29 a2+ x35
x30 a2+ x36
x31 a2+ x37
x32 a2+ x38 z2+ x34 a2- x33 z2- x32
x39 a2+ x51 z2+ x47 a2- x43 z2- x39
x40 a2+ x52 z2+ x48 a2- x44 z2- x40
x41 a2+ x53 z2+ x49 a2- x45 z2- x41
x42 a2+ x54 z2+ x50 a2- x46 z2- x42

x51 a3+ x54 z3+ x53 a3- x52 z3- x51
x47 a3+ x50 z3+ x49 a3- x48 z3- x47
x43 a3+ x46 z3+ x45 a3- x44 z3- x43
x39 a3+ x42 z3+ x41 a3- x40 z3- x39
x35 a3+ x38 z3+ x37 a3- x36 z3- x35
x29 a3+ x32 z3+ x31 a3- x30 z3- x29
x25 a3+ x28 z3+ x27 a3- x26 z3- x25
x23 a3+ x24
x19 a3+ x22 z3+ x21 a3- x20 z3- x19
x15 a3+ x18 z3+ x17 a3- x16 z3- x15
x11 a3+ x14 z3+ x13 a3- x12 z3- x11
x09 a3+ x10
x05 a3+ x08 z3+ x07 a3- x06 z3- x05
x01 a3+ x04 z3+ x03 a3- x02 z3- x01
.marking  x01
.end
```

Figure 5.7: Specification of an arbiter with 2 conflict points associated with exclusion relation of 2 out of 3. The definition regions of both conflict points have the same input interfaces. The state graph description is too large to have a legible pictorial representation. Thus we give: (a) its Petri net description; (b) its state graph description in the ASTG format, used by *petrify*. To understand the latter description note that, in the ".state graph" section, each line represents a path, where symbols starting with an "x" represent states. The initial state is $x01$.

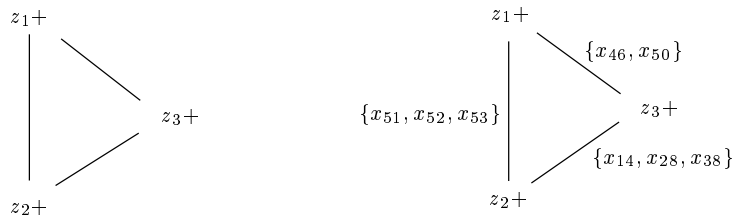


Figure 5.8: Non-persistence graph for the specification depicted in figure 5.7.

appears. The non-persistence graph, with edge labels, of the system depicted in figure 5.7 is depicted in figure 5.8.b.

Let p be a conflict point caused by n concurrent regions in an exclusion relation of k out of n and let $\langle I_i, O_i \rangle$ and $\langle I_j, O_j \rangle$ be two of the n concurrent regions. Let s be a state with a non-persistence between events e_i and e_j, such that $e_i \in I_i$ and $e_j \in I_j$. How can we know that s is covered by p? Since we are behind an exclusion relation of k out of n, in order to exist a non-persistence in s, s must be inside $k-1$ of the concurrent regions, others than $\langle I_i, O_i \rangle$ and $\langle I_j, O_j \rangle$. Thus, if s is not inside $k-1$ of the concurrent regions, the non-persistence associated with it will remain after the conflict point is controlled by a genex.

For each conflict point selected to be controlled, we can remove from the labels of the non-persistence graph the states (non-persistences) covered by it. A given complete component (subgraph) of the non-persistence graph becomes controlled only when all its edges are labeled with the empty set. Let use the system in figure 5.7, with the non-persistence graph depicted in figure 5.8, to illustrate this covering analysis. Remember we have detected two solutions for conflict points associated to the complete component of the non-persistence graph. Considering the first solution we have that $x_{14}, x_{28}, x_{38} \in \langle I_1, O_1 \rangle$, $x_{50} \in \langle I_2, O_2 \rangle$ and $x_{52}, x_{53} \in \langle I_3, O_3 \rangle$, but $x_{46} \notin \langle I_2, O_2 \rangle$ and $x_{51} \notin \langle I_3, O_3 \rangle$ and so x_{46} and x_{51} are not covered by this conflict point. Using only the second solution we have that $x_{38} \in \langle I_1, O_1' \rangle$, $x_{46}, x_{50} \in \langle I_2, O_2' \rangle$ and $x_{51}, x_{52}, x_{53} \in \langle I_3, O_3' \rangle$, but $x_{14}, x_{51} \notin \langle I_1, O_1' \rangle$ and so these two states are not covered by the second conflict point. The sets of states not covered are disjoint, and so using two genexes 3×2 all non-persistences are controlled.

In section 4.3.1 we have referred that when a mutex is used to control a conflict, different options exist for the insertion of the release and completion events. This is in agreement with the current covering

analysis. Let a conflict point be associated with n concurrent regions in an exclusion relation of 1 out of n. A non-persistence state associated with 2 of the n concurrent regions is covered if it is inside 0 (zero) of the other concurrent regions. This is true whatever the output interfaces of the considered concurrent regions are.

5.4 Genex Insertion

Once a genex has been selected to control an identified conflict point, the state graph specification has to be transformed to reflect the use of the genex. This transformation is determined by procedures 4.15 and 4.16. The insertion of each channel of the genex is based on the concurrent region associated to the conflict point. Let G be the state graph to be transformed; let $\langle I_i, O_i \rangle$ be the region associated with channel i of the genex. let r_i and g_i be the input and output signals for channel i of the genex. Following procedure 4.15, let α_i be the minimum region having I_i as the output interface. We are developing next an expression to "insert r_i+ in G such that it appears after the α_i events but before the I_i events".

Region-Based Event Insertion

Let G be a state graph and R a region on it. R satisfies conditions of theorem 3.28 and so it is a SIP-set. The insertion of a new event e' in G by R based on definition 3.25 produces a transition system G' where all transitions labeled with events from $R\bullet$, the output interface of R, are delayed by e'. This allows to represent the event insertion in product form.

Definition 5.3 (s-factor)
Let I and O be two sets of events, such that $I \cap O = \emptyset$; let k be an integer with value 0 or 1. The transition system $F = \langle X, E, \Theta, x_{in} \rangle$ where

$$
\begin{aligned}
X &= \{x_0, x_1\} \\
E &= I \cup O \quad , \quad I \cap O = \emptyset \\
\Theta &= \{\langle x_0, e, x_1 \rangle \mid e \in I\} \cup \{\langle x_1, e, x_0 \rangle \mid e \in O\}
\end{aligned}
$$

$$x_{in} \quad = \quad \begin{cases} x_0 & k = 0 \\ x_1 & k = 1 \end{cases}$$

is called an s-factor*, and is denoted by* $\mathbf{sfactor}(k, I, O)$.

Let G be a transition system, s_{in} its initial state, $R = \langle I, O \rangle$ a region on it, defined by its interface, and e' a new event.

Definition 5.4 (region-based event insertion)

The region-based event insertion *of e' in G by R transforms G into another transition system G', given by expression*

$$G' = \mathbf{reach}\,(G \times \mathbf{sfactor}(k, I, \{e'\}) \times \mathbf{sfactor}(0, \{e'\}, O))$$

where k is equal zero if $s_{in} \notin R$, and 1 otherwise.

Theorem 5.5 (equivalence of event insertions)

Up to reachability, the insertions of e' in G by R based on definitions 3.25 and 5.4 are isomorphic.

Proof

Let $G' = \langle S', E', \Theta', {s'}_{in} \rangle$ be the transition system obtained after insertion by definition 3.25. Then $S' = S \cup \imath(R)$, $E' = E \cup \{e'\}$, and ${s'}_{in} = s_{in}$. Set Θ' has 4 pieces which can be rewritten as follows

$$\begin{aligned} \Theta'_A &= \{\langle s, e, s' \rangle \mid \langle s, e, s' \rangle \in \Theta \ \wedge \ (s \notin r \ \vee \ s' \in r)\} \\ &= \{\langle s, e, s' \rangle \mid \langle s, e, s' \rangle \in \Theta \ \wedge \ e \in E - O\} \end{aligned}$$

because R is a region, whose interface we assume as being $\langle I, O \rangle$.

$$\begin{aligned} \Theta'_B &= \{\langle \imath(s), e, \imath(s') \rangle \mid \langle s, e, s' \rangle \in \Theta \ \wedge \ s, s' \in R\} \\ \Theta'_C &= \{\langle \imath(s), e, s' \rangle \mid \langle s, e, s' \rangle \in \Theta \ \wedge \ s \in R \ \wedge \ s' \notin R\} \\ &= \{\langle \imath(s), e, s' \rangle \mid \langle s, e, s' \rangle \in \Theta \ \wedge \ e \in O\} \end{aligned}$$

again because R is a region. Finally

$$\Theta'_D = \{\langle s, e', \imath(s) \rangle \mid s \in R\}$$

let $G'' = \langle S'', E'', \Theta'', {s''}_{in} \rangle$ be the transition system obtained after insertion by definition 5.4. Let $X = \{x_0, x_1\}$ be the set of states of the first s-factor in definition, and $Y = \{y_0, y_1\}$ the

set of states of the second. By the definition of asynchronous product

$$\begin{aligned} S'' &= S \times X \times Y \\ &= ((S - R) \cup R) \times \{x_0, x_1\} \times \{y_0, y_1\} \\ &= R_{000} \cup R_{100} \cup R_{010} \cup R_{110} \cup R_{001} \cup R_{101} \cup R_{011} \cup R_{111} \end{aligned}$$

$$E'' = E \cup \{e'\} = E'$$

and, taken into account that set of events I is common to G and the first s-factor, set of events O is common to G and the second s-factor, event e' is common to both s-factors, and set of events $E - I - O$ only appear in G,

$$\begin{aligned} \Theta'' &= \{\langle\langle s, x, y\rangle, e, \langle s', x, y\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in E - I - O\} \\ &\cup \{\langle\langle s, x_0, y\rangle, e, \langle s', x_1, y\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in I\} \\ &\cup \{\langle\langle s, x, y_1\rangle, e, \langle s', x, y_0\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in O\} \\ &\cup \{\langle\langle s, x_1, y_0\rangle, e', \langle s, x_0, y_1\rangle\rangle \mid s \in R\} \end{aligned}$$

Analyzing set Θ'' in terms of flow of transitions between the 8 parts of S'', we get that the possible flows are: $R_{000} \to R_{110}$, $R_{001} \to R_{111}$, $R_{101} \to R_{000}$, $R_{111} \to R_{010}$, $R_{010} \to R_{001}$, and $R_{110} \to R_{101}$. Initial state ${s''}_{in}$ belongs to R_{000} or R_{110} depending on set s_{in} belongs to, either $S - R$ or R. In any case, sets of states R_{001}, R_{010}, R_{011}, R_{100} and R_{111} are not reachable from the initial state.

Let $G''' = \langle S''', E''', \Theta''', {s'''}_{in}\rangle = G''[R_{000} \cup R_{110} \cup R_{101}]$. Then,

$$\begin{aligned} S''' &= R_{000} \cup R_{110} \cup R_{101} \\ &= (S - R) \times \langle x_0, y_0\rangle \cup R \times \langle x_1, y_0\rangle \cup R \times \langle x_0, y_1\rangle \end{aligned}$$

$$E''' = E''$$

$$\Theta''' = \Theta'''_A \cup \Theta'''_B \cup \Theta'''_C \cup \Theta'''_D \cup \Theta'''_E \cup \Theta'''_F$$

being

$$\Theta'''_A = \{\langle\langle s, x_0, y_0\rangle, e, \langle s', x_0, y_0\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in E - I - O \ \wedge \ s, s' \in S - R\}$$

$$\Theta'''_B = \{\langle\langle s, x_1, y_0\rangle, e, \langle s', x_1, y_0\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in E - I - O \ \wedge \ s, s' \in R\}$$

$$\Theta'''_C = \{\langle\langle s, x_0, y_1\rangle, e, \langle s', x_0, y_1\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in E - I - O \ \wedge \ s, s' \in R\}$$

$$\Theta'''_D = \{\langle\langle s, x_0, y_0\rangle, e, \langle s', x_1, y_0\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in I\}$$

$$\Theta'''_E = \{\langle\langle s, x_0, y_1\rangle, e, \langle s', x_0, y_0\rangle\rangle \mid \langle s, e, s'\rangle \in \Theta \ \wedge \ e \in O\}$$

$$\Theta'''_F = \{\langle\langle s, x_1, y_0\rangle, e, \langle s, x_0, y_1\rangle\rangle \mid s \in R\}$$

Let define function $\sigma : S' \to S'''$ *such that*

$$\sigma(s') = \begin{cases} \langle s', x_0, y_0\rangle & s' \in S - R \\ \langle s', x_1, y_0\rangle & s' \in R \\ \langle s', x_0, y_1\rangle & s' \in R' \end{cases}$$

Function σ *is clearly a bijection. Let define function* $\tau : \Theta' \to \Theta'''$ *such that*

$$\tau(\langle s, e, s'\rangle = \langle \sigma(s), e, \sigma(s')\rangle$$

Since σ *is injective,* τ *also is. Determining the image of* Θ' *we get*

$$\begin{aligned} \tau(\Theta'_A) &= \Theta'''_A \cup \Theta'''_B \cup \Theta'''_D \\ \tau(\Theta'_B) &= \Theta'''_C \\ \tau(\Theta'_C) &= \Theta'''_E \\ \tau(\Theta'_D) &= \Theta'''_F \end{aligned}$$

Thus τ *is also surjective and hence the pair* $\langle\sigma, \tau\rangle$ *is an isomorphism from* G' *into* G'''*. Finally,*

$$\mathbf{reach}(G') \cong \mathbf{reach}(G''') = \mathbf{reach}(G'')$$

Channel Insertion

Let us return to the insertion of r_i+ in G such that it appears after the α_i events but before the I_i events. $\langle\alpha_i, I_i\rangle$ is a region and thus the insertion can be done by the product

$$G^1 = \mathbf{reach}(G \times \mathbf{sfactor}(k_i, \alpha_i, \{r_i+\}) \times \mathbf{sfactor}(0, \{r_i+\}, I_i))$$

where k_i is equal 0 if $s_{in} \notin \langle\alpha_i, I_i\rangle$, and 1 otherwise, s_{in} being the initial state of G. The second step of procedure 4.15, "insertion of g_i+ such that it appears after event r_i+ but before events from I_i", can be done by the product

$$G^2 = \mathbf{reach}(G^1 \times \mathbf{sfactor}(0, \{r_i+\}, \{g_i+\}) \times \mathbf{sfactor}(0, \{g_i+\}, I_i))$$

To implement step 3 of procedure 4.15 we need the minimum region having O_i as the input interface. Let $\langle O_i, \beta_i\rangle$ be such a region. To "insert r_i- in G^2 such that it appears after the events from O_i but before the events from β_i", we can use the following product

$$G^3 = \mathbf{reach}(G^2 \times \mathbf{sfactor}(m_i, O_i, \{r_i-\}) \times \mathbf{sfactor}(0, \{r_i-\}, \beta_i))$$

where m_i is equal 0 if $s_{in} \notin \langle O_i, \beta_i \rangle$, and 1 otherwise. Finally, to "insert g_i- such that it appears after event r_i- but before events from β_i" (step 4 of procedure 4.15), we can use the product

$$G^4 = \mathbf{reach}(G^3 \times \mathbf{sfactor}(0, \{r_i-\}, \{g_i-\}) \times \mathbf{sfactor}(0, \{g_i-\}, \beta_i))$$

In order to implement step 5 of procedure 4.15, let assume that $\alpha_i = O_i$. If we determine β_i before inserting r_i+ and g_i+, we will get $\beta_i = I_i$. Indeed, if the minimum region having I_i as the output interface has O_i as the input interface, then the minimum region having O_i as the input interface has I_i as the output interface. But after step 1 of procedure 4.15 the minimum region having O_i as the input interface is $\langle O_i, \{r_i+\}$. Thus after steps 3 and 4 we have introduced the sequence $\langle r_1-, g_1-, r_1+, g_1+\rangle$ between O_i and I_i. This will result in the introduction of a CSC conflict. This conflict can be easily removed by inserting a new signal, with, for instance, the positive transition between g_i- and r_i+ and the negative transition between I_i and O_i. Assuming csc_i is the new added signal, this insertion can be done with the product

$$\begin{aligned} G^5 = &\mathbf{reach}(G^4 \times \mathbf{sfactor}(0, \{g_i-\}, \{csc_i+\}) \times \mathbf{sfactor}(0, \{csc_i+\}, \{r_i+\}) \\ &\times \mathbf{sfactor}(n_i, I_i, \{csc_i-\}) \times \mathbf{sfactor}(0, \{csc_i-\}, O_i) \end{aligned}$$

where n_i takes the value 1 if $s_{in} \in \langle I_i, O_i \rangle$, and 0 otherwise.

Correctness of Signal Insertion

Clearly, G^1, G^2 and G^3 are not valid state graphs. Actually they do not need to be so, since they are intermediate values. But both G^4 and G^5 represent transformed state graphs, and so they have to be state graphs. Can we guarantee that?

Theorem 5.6 (region-based signal insertion)
Let G be a state graph, s_{in} its initial state and $R_1 = \langle I_1, O_1 \rangle$ and $R_2 = \langle I_2, O_2 \rangle$ two regions on G satisfying the following conditions:

1. $R_1 \cap R_2 = \emptyset$;
2. *if $O_1 \neq I_2$, then $\langle O_1, I_2 \rangle$ is a region; and*
3. *if $O_2 \neq I_1$, then $\langle O_2, I_1 \rangle$ is a region.*

Let v be a new signal. The transition system given by equation

$$G' = \mathbf{reach}(G \times \mathbf{sfactor}(k_1, I_1, \{v+\}) \times \mathbf{sfactor}(0, \{v+\}, O_1) \\ \times \mathbf{sfactor}(k_2, I_2, \{v-\}) \times \mathbf{sfactor}(0, \{v-\}, O_2)),$$

where k_1 is 1 if $s_{in} \in \langle I_1, O_1 \rangle$ and 0 otherwise and k_2 is 1 if $s_{in} \in \langle I_2, O_2 \rangle$ and 0 otherwise, is a (consistent) state graph.

Proof

Let $S^+ = R_1$ and $S^- = R_2$. If $I_1 \neq O_2$ let $S^0 = \langle O_2, I_1 \rangle$; otherwise let $S^0 = \emptyset$. If $I_2 \neq O_1$ let $S^1 = \langle O_1, I_2 \rangle$; otherwise let $S^1 = \emptyset$. I-Partition $I = \langle S^0, S^+, S^1, S^- \rangle$ conforms to theorem 3.29.

The equation for G' can be rewritten as

$$G'' = \mathbf{reach}(G \times \mathbf{sfactor}(k_1, I_1, \{v+\}) \times \mathbf{sfactor}(0, \{v+\}, O_1))$$
$$G' = \mathbf{reach}(G'' \times \mathbf{sfactor}(k_2, I_2, \{v-\}) \times \mathbf{sfactor}(0, \{v-\}, O_2))$$

The former of these equations represents the insertion of $v+$ in G by S^+ (see theorem 5.5, definition 3.25 and section 3.7). The latter represents the insertion of $v-$ in G' by S^-. Thus, G' is a (consistent) state graph.

First we prove that if G^4 is a state graph, then G^5 also is. Assume G^4 is a state graph. On G^4 regions $\langle \{g_i-\}, \{r_i+\} \rangle$ and $\langle I_i, O_i \rangle$ are disjoint and $\langle \{r_i+\}, I_i \rangle$ and $\langle O_i, \{g_i-\} \rangle$ are regions. Thus, by theorem 5.6, G^5 is a (consistent) state graph.

Let now prove that G^4 also is a state graph. By properties 3.13 and 3.14, up to isomorphism, equation for G^4 can be rewritten as

$$G^r = \mathbf{reach}(G \times \mathbf{sfactor}(k_i, \alpha_i, \{r_i+\}) \times \mathbf{sfactor}(0, \{r_i+\}, I_i) \\ \times \mathbf{sfactor}(m_i, O_i, \{r_i-\}) \times \mathbf{sfactor}(0, \{r_i-\}, \beta_i))$$
$$G^4 \cong \mathbf{reach}(G^r \times \mathbf{sfactor}(0, \{r_i+\}, \{g_i+\}) \times \mathbf{sfactor}(0, \{g_i+\}, I_i) \\ \times \mathbf{sfactor}(0, \{r_i-\}, \{g_i-\}) \times \mathbf{sfactor}(0, \{g_i-\}, \beta_i))$$

If $\alpha_i \neq O_i$, then regions $\langle \alpha_i, I_i \rangle$ and $\langle O_i, \beta_i \rangle$ are disjoint. If $\alpha_i = O_i$, then $\beta_i = \{r_i+\}$ and again $\langle \alpha_i, I_i \rangle$ and $\langle O_i, \beta_i \rangle$ are disjoint regions. If $I_i \neq O_i$, then $\langle I_i, O_i \rangle$ is a region. Finally, if $\alpha_i \neq \beta_i$, then $\langle \beta_i, \alpha_i \rangle$ is also a region. Thus, by theorem 5.6, G^r is a state graph.

In G^r, regions $\langle \alpha_i, I_i \rangle$ and $\langle O_i, \beta_i \rangle$ still are disjoint; $\langle \{r_i+\}, I_i \rangle$ is a sub-region of $\langle \alpha_i, I_i \rangle$ and $\langle \{r_i-\}, \beta_i \rangle$ is a sub-region of $\langle O_i, \beta_i \rangle$ and so they are disjoint. Either $\langle I_i, O_i \rangle$ is a region or $I_i = O_i$. Then any way $\langle I_i, \{r_i-\} \rangle$ is a region. If $\alpha_i \neq I_i$, either $\langle \beta_i, \alpha_i \rangle$ is a region or $\beta_i = \alpha_i$, and so $\langle \beta_i, \{r_i+\} \rangle$ is a region. If $\alpha_i = I_i$, $\beta_i = \{r_i+\}$. Conditions of theorem 5.6 are satisfied and thus G^4 is a state graph.

From G to G^4 or G^5, the transformation is done by successive products with *s-factors*. Since the product of transition systems is associative, the insertion of genex channel i into G can be defined by the product

$$G' = G \times F_i$$

where G' represents the transformed state graph after insertion of channel i and F_i represents the insertion transformation, that is, F_i represents the product of the several *s-factors*.

Genex Insertion

The channel insertion procedure (procedure 4.15) has to be applied to all channels of the genex, as determined by step 1 of procedure 4.16. In product form this can be written as

$$G' = G \times F_1 \times F_2 \times \cdots \times F_n$$

where G' represents the transformed state graph after insertion of all channels and F_i, for $i = 1, 2, \cdots, n$, represent the n channels of the genex. Up to isomorphism the product of transition systems is commutative. Thus, the order of insertion of the n channels is irrelevant.

Elimination of Extra States

After step 1 of procedure 4.16 is concluded, the transformed state graph hasn't taken into account the exclusion relation among the grant signals of the genex. All states that correspond to having more than k grant signals simultaneously at 1, can be removed, since they are unreachable. This elimination, which corresponds to step 2 of procedure 4.16, can be accomplished by the product of

the transformed state graph with the state graph description of the genex behavior. Assuming X represents this state graph, the total transformation needed to insert a genex is given by the product

$$G' = G \times F_1 \times F_2 \times \cdots \times F_n \times X$$

An Example

Let us return to the example of figures 5.7 and 5.8, discussed in section 5.3. As we have mentioned it has two conflict points, in the form of exclusion relations of 2 out of 3. The concurrent regions are

$$R_1 = \langle I_1, O_1 \rangle = \langle \{z_1+\}, \{a_1+\} \rangle$$
$$R_2 = \langle I_2, O_2 \rangle = \langle \{z_2+\}, \{a_2-\} \rangle$$
$$R_3 = \langle I_3, O_3 \rangle = \langle \{z_3+\}, \{z_3-\} \rangle$$

for one of the conflicts and

$$R_4 = \langle I_1, O'_1 \rangle = \langle \{z_1+\}, \{a_1-\} \rangle$$
$$R_5 = \langle I_2, O'_2 \rangle = \langle \{z_2+\}, \{z_2-\} \rangle$$
$$R_6 = \langle I_3, O'_3 \rangle = \langle \{z_3+\}, \{a_3+\} \rangle$$

for the other. Let control the first using a genex 3×2, being r_1, r_2 and r_3 the input signals and g_1, g_2 and g_3 the output signals. Determining the minimum regions having I_1, I_2 and I_3 as output interfaces we obtain for the input interfaces respectively

$$\alpha_1 = \{a_1+\}, \quad \alpha_2 = \{a_2+\} \quad \text{and} \quad \alpha_3 = \{a_3+\}.$$

Determining the minimum regions having O_1, O_2 and O_3 as input interfaces, we obtain for the output interfaces respectively

$$\beta_1 = \{r_1+\}, \quad \beta_2 = \{z_2-\} \quad \text{and} \quad \beta_3 = \{a_3+\}.$$

Note that $\alpha_1 = O_1$, and thus we have $\beta_3 = \{r_1+\}$ and we also have to insert a new signal to resolve the CSC conflict appearing during insertion of channel 1 of the genex.

After genex insertion we obtain a state graph with 13 signals (7 new), 572 states and 2 non-persistence conflicts, corresponding to the 2 states, x_{46} and x_{51}, we already knew were not covered by the chosen conflict point.

5.5 Input signals

The overall specification after transformation (transformed state graph plus arbiters) should preserve speed independence, which includes the interaction with the environment. If in a given specification an input signal transition appears first after some output signal transition, we can assume that somehow the environment is sensing the circuit and guarantees that the input signal transition does not appear before. In the transformation process presented in the previous section, sets of events I_i and β_i are delayed, the former by the request to and the grant from the arbiter and the latter by the release and completion events. If any of these sets of events contain transitions on input signals then we are delaying them, which should be overcome.

On the other side and for the sake of simplicity we want to use the same transformation for all types of events. Thus we propose a pre-treatment of the input signals in order to transfer to some internal signal any non-persistence relation they are involved with.

If the environment is managing to fire a signal transition (an input signal transition from the circuit point of view) the circuit can not avoid it. But it could eventually delay its effect on the circuit. The input signals are going from the environment to the circuit through wires. Consider that under the assumption that the delay in one of these wires is not negligible the circuit works correctly. Thus we can consider that the ends of these wires are different signals and transform the specification in order to take this into account. In this way conflicts involving the input signal are eventually transferred to the signal associated to the final end of the wire, which is an internal signal from the circuit point of view.

Let G be a state graph and v an input signal involved in some non-persistence we want to control. We start renaming v, let say as v_f, and reclassifying it as an internal signal. Next, we insert the new input signal v such that it directly precedes v_f, that is, $v+$ and $v-$ are inserted based on regions having $\{v_f+\}$ and $\{v_f-\}$ as the output interfaces. To obtain the input interfaces of the insertion regions we have opted to use minimum regions. Let π and μ represent the input interfaces of the minimum regions having respectively $\{v_f+\}$ and $\{v_f-\}$ as the output interfaces. After renaming v as v_f, the insertion of the new v can be computed by the product

$$G' = \mathbf{reach}(G'' \times \mathbf{sfactor}(0, \pi, \{v+\}) \times \mathbf{sfactor}(k_\pi, \{v+\}, \{v_f+\}) \\ \times \mathbf{sfactor}(0, \mu, \{v-\}) \times \mathbf{sfactor}(k_\mu, \{v-\}, \{v_f-\})$$

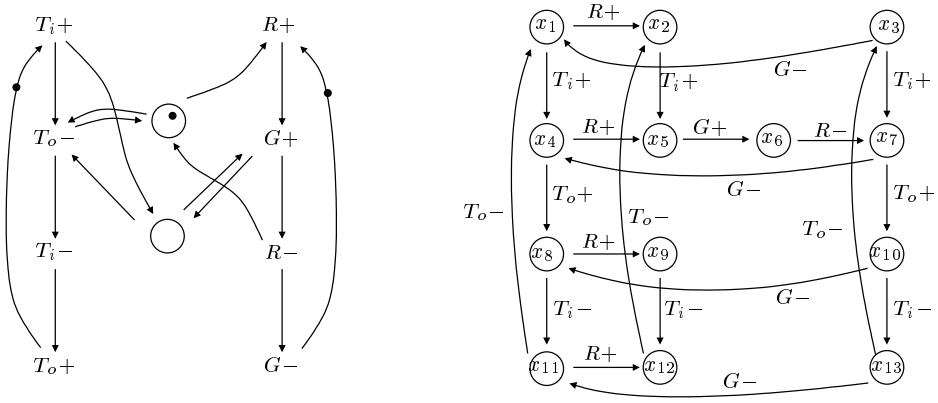


Figure 5.9: A specification with an asymmetric non-persistence involving an input signal transition: (a) STG specification; (b) corresponding reachability (state) graph, where x_1 is the initial state and the non-persistence is associated with state x_4.

where

$$k_\pi = \begin{cases} 1 & s_{in} \in \langle \pi, \{v_f+\} \rangle \\ 0 & \text{otherwise} \end{cases}$$

$$k_\mu = \begin{cases} 1 & s_{in} \in \langle \mu, \{v_f-\} \rangle \\ 0 & \text{otherwise} \end{cases}$$

Let illustrate with an example. In [53] a number of arbiters, so-called token-ring arbiters, are presented. In the specification depicted in figure 5.9, the STG description was borrowed from there. Signals T_i and R are inputs while T_o and G are outputs. It is clear, even in the STG domain, that there is an asymmetric non-persistence, with the output signal transition T_o+ being disabled by input signal transition $R+$. The corresponding reachability (state) graph is depicted in figure 5.9.b, where x_1 is the initial state and the non-persistence is associated with state x_4.

Let G represent the state graph of the token-ring arbiter in the figure and let control its conflict point, that is, the non-persistence in state x_4. Since R is an input signal we must start pre-treating it. First of all we rename R as R_f and reclassify it as internal. The input interfaces of the smallest regions with

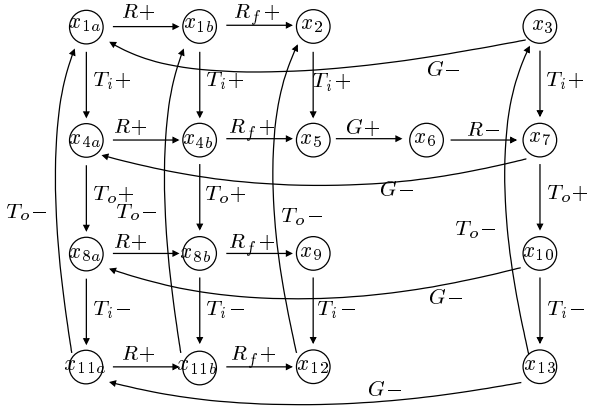


Figure 5.10: Transformed state graph obtained from the one in figure 5.9 in order to transfer the non-persistence involving an input signal to a non-persistence between output signals. In this state graph there is a symmetric non-persistence between signal transitions T_o+ and $G+$ at state x_{4b}.

output interfaces $\{R_f+\}$ and $\{R_f-\}$ are respectively

$$\pi = \{G-\}, \quad \mu = \{G+\}$$

with the initial state of G belonging to region $\langle\{G-\},\{R_f+\}\rangle$, which means that $k_\pi = 1$ and $k_\mu = 0$. The pre-transformation is thus given by the product

$$\begin{aligned} G' = \mathbf{reach}(G'' \times \mathbf{sfactor}(0,\{G-\},\{R+\}) \times \mathbf{sfactor}(1,\{R+\},\{R_f+\}) \\ \times \mathbf{sfactor}(0,\{G+\},\{R-\}) \times \mathbf{sfactor}(0,\{R-\},\{R_f-\}) \end{aligned}$$

where G'' represents the state graph obtained from G after renaming R as R_f. G' is depicted in figure 5.10. The original non-persistence was transformed into a symmetric non-persistence between output signal transitions T_o+ and $G+$.

5.6 TSF Toolset

The first purpose of this chapter was to introduce an automated method to make synthesis of asynchronous circuits from specifications with conflicts, in the form of non-persistences. The method takes a state graph specification with conflicts and delivers a set of special arbiters plus a conflict-free state graph specification. This state graph must be suitable to be synthesized using existing

speed-independent synthesis tools, like *petrify*[22, 20]. We have developed a set of small tools which implement different steps of the transformation process. It is called the *TSF toolset*[3], and is composed of a data structure, a support library and a set of command line applications.

The main operation of the transformation process is the product of transition systems, which most of the time are not state graphs. So, tools must be supported by a data model to store transition systems. This data model appears in two ways: as a text file format and as a memory object model.

The text file format represents the interface with the user. We have defined a model where events, states and transitions are explicitly represented. The object model is based on the notion of transition system folder. This is basically a set of transition systems sharing a global set of events and a set of primitive sets of states. The idea behind this folder is to make easy the product of transition systems. From one side, we need to synchronize in common events. The global set of events simplifies this task. On the other side, the set of states of the product is a subset of the Cartesian product of the operand sets of states. Thus, states on transition systems are assumed to be tuples of primitives states. This allows to keep the state connection between original and transformed transition systems.

On top of the object model a library of functions was developed. On a lower layer these functions fulfill a set of basic operations, like event, state and transition operations, load and print operations, and so on. On a upper layer they fulfill more complex operations, like the product, parsing and loading, region extraction, etc.

A set of command line applications was developed. Some are built on top of the library functions. Others were written as Bourne shell scripts and are used to generate simple transition systems, like s-factors. These tools were used to obtain the data of the several examples presented in this thesis.

5.6.1 TSF File Format

A specific text file format, called TSF file format, was defined to interact with the tool set. Sets of events, states and transitions are given explicitly in this format. The grammar of the TSF file format is depicted in figure 5.11. A $*$ after a symbol represents 0 or more occurrences of it. A $+$ represents 1 or more occurrences. Note that a transition system can be composed of a single state, with no events nor transitions. Figure 5.12 shows a transition system description in the TSF file format.

[3] TSF stands for transition system folder.

ts-description	=	**model name events states transitions initial-state end**
model	=	*".model" "transition system"*
name	=	*".name"* string
events	=	*".events"* **(event)**$*$
event	=	string
states	=	*".states"* **(state)**$+$
state	=	string
transitions	=	*".transitions"* **(transition)**$*$
transition	=	*"<"* state *","* event *","* state *">"*
initial-state	=	*".initial state"* **state**
end	=	*".end"*

Figure 5.11: The grammar of the TSF file format.

The TSF file format does not hold information about signal type. Actually, transition systems only have events, independently of their nature. Thus converting a state graph description to the TSF file format, there is a loss of information, which makes it impossible to revert. To avoid this, we use another text file format just to keep signal type information. It is defined as a set of statements, each one composed of one of the keywords `.inputs`, `.outputs` and `.internal` followed by a list of signals of that type. Any number of statements can exist. The same signal can be reclassified several times, the last being the one effective. This makes it easy to add new signals and reclassify existing ones.

5.6.2 TSF Object Model

A simple overview of the TSF object model is given in figure 5.13, where each box represents a different data type. The main data type is the *transition system folder* which represents a set of *transition systems*. Each transition system is composed, as expected, by a set of events, a set of states and a set of transitions. We define two types of events and two types of states. Data type *event* represents an event independently of the transition system where it occurs, while data type *ts-event* is an event in a given transition system. The latter is a sort of instantiation of the former. Data type *bare-state* represents a state in a transition system indicated by the user. Internally a *state* is a tuple

```
#
# Transition System of the a simple arbiter
# This is an arbiter with a conflict point between output signal
# transitions used for illustration purposes.
#
.model transition system
.name s-arbiter
.events # 8 events
    a1+ a1- a2+ a2- b1+ b1- b2+ b2-

.states # 11 states
    x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11

.transitions # 17 transitions
 <x1,a1+,x2>
 <x3,a1+,x4> <x4,b1+,x5> <x5,a1-,x6> <x6,b1-,x3>
 <x9,b1-,x7> <x7,a1+,x8>
 <x10,a1+,x11>

 <x1,a2+,x3> <x3,b2+,x7> <x7,a2-,x10> <x10,b2-,x1>
 <x2,a2+,x4> <x4,b2+,x8> <x8,a2-,x11> <x11,b2-,x2>
 <x6,b2+,x9>

.initial state x1
.end
```

Figure 5.12: TSF description of the state graph depicted in figure 4.17.

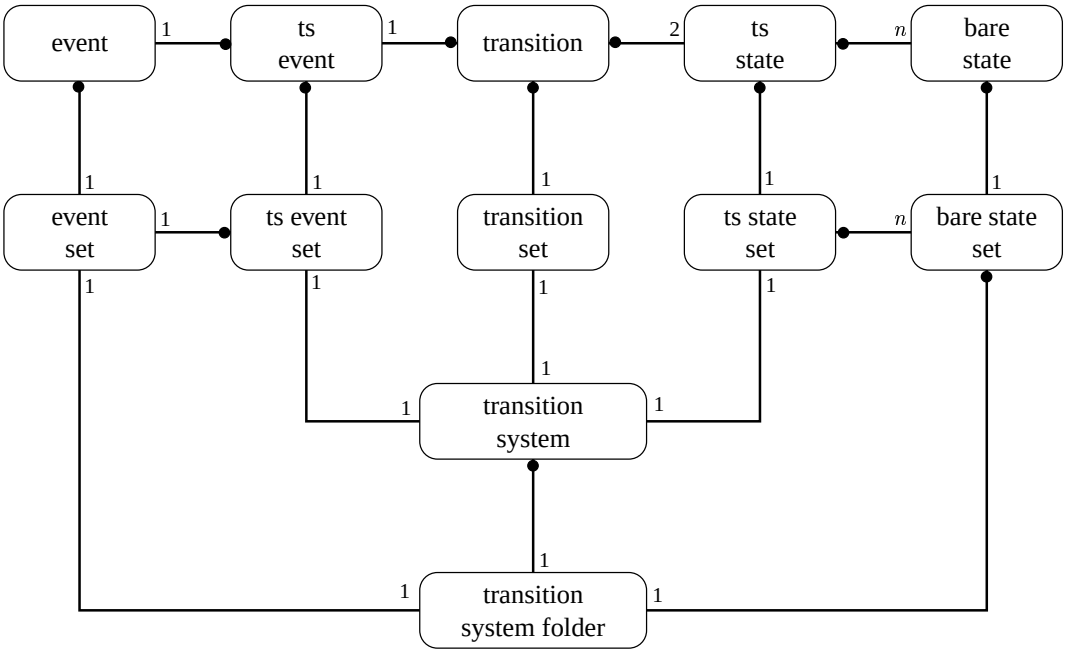


Figure 5.13: The TSF Object Model.

of one or more bare states, each from a different bare state set. This makes it easy to correlate states in the original and the transformed transition systems. It is particularly useful during the product operation, because we can easily keep a connection between operands and result.

5.6.3 TSF Library

Using the data model explained in the previous section a library of functions was developed in the C programming language. It is referred to as the *TSF library* and two layers on it can be considered. In a lower layer there are basic functions which manipulate data types other than transition system and transition system folder. In an upper layer there are more complex operations, like parsing, loading, product, projection, region searching. The TSF library is the foundation for several of the tools described in next section.

5.6.4 TSF Tools

At present available tools take the form of a set of command line applications, which can be divided into two groups. In one group, there are a set of tools developed on top of the TSF library. The other group of tools were written as Bourne shell scripts and are used to generate simple transition systems used as factors in products and to cluster several operations into a single one. Most of them are intended as temporary versions of future, more efficient ones. A list of some of the available tools follows, along with the synopsis and a summary description.

sg2ts

TYPE: C program

SYNOPSIS: `sg2ts [-i sg-file] [-o ts-file] [-t t-file]`

DESCRIPTION: Converts a state graph description from petrify state graph format to TSF file format. By default input is read from standard input and output sent to standard output. Modifiers `-i` switches input to file `sg-file`, `-o` switches output file to `ts-file` and `-t` creates the file `t-file` with types for signals. Note that TSF does not have information about signals, only about events. This file can be used afterwards to convert from TSF format to petrify SG format.

ts2sg

TYPE: C program.

SYNOPSIS: `ts2sg [-i ts-file] [-o sg-file] [-t t-file]`

DESCRIPTION: Converts a state graph description form TSF file format to petrify SG format. By default input is read from standard input, output sent to standard output and all signals are given type internal. Modifiers `-i` switches input to file `ts-file`, `-o` switches output file to `sg-file` and `-t` uses information on `t-file` to give signals the correct type.

ts-trivial

TYPE: Bourne shell script.

SYNOPSIS: `ts-trivial [ -name name ]`

DESCRIPTION: Generates the TSF description of a trivial transition system named `name`, if option `-name` is given, or “triv-#”, where # is a random number, otherwise. A trivial TS has a single state, no events and no transitions. It is used as an auxiliary TS in other operations.

s-factor

TYPE: Bourne shell script.

SYNOPSIS: `s-factor [-name name] k I O`

DESCRIPTION: Generates the TSF description of the $\mathbf{sfactor}$(k,I,O) (see definition 5.3). The generated transition system is named `name`, if option `-name` is given, or "sf-#", where # is a random number, otherwise.

ts-prod

TYPE: C program.

SYNOPSIS: `ts-prod [-name name] opnd1 opnd2 [outfile]`

DESCRIPTION: Computes the product between transition systems `opnd1` and `opnd2`. TSF description of the product is sent to `outfile`, if given, or standard output, otherwise. The generated transition system is named `name`, if option `-name` is given, or "ts-#", where # is a random number, otherwise.

ts-mprod

TYPE: Bourne shell script..

SYNOPSIS: `ts-mprod [-name name] opnd1 [opnd2 [...]]`

DESCRIPTION: Computes the product of all given transition systems. TSF description of the product is sent to standard output. It is named `name`, if option `-name` is given, or "ts-#", where # is a random number, otherwise.

ts-detect-np

TYPE: C program

SYNOPSIS: `ts-detect-np [infile]`

DESCRIPTION: Prints on standard output all disabling situations on the given transition system (the standard input as default).

ts-regions

TYPE: C program

SYNOPSIS: `ts-regions [-ts infile] [-b back-set] [fore-pattern]`

DESCRIPTION: Prints, on standard output, the interfaces of all regions of the given transition system (default standard input) satisfying the given initial scenario. Initially, events are distributed among 8 different sets, to know, I (input), O (output), NC (not-cross), IO (input-output, associated to empty regions), UONC (output or not-cross), UINC (input or not-cross),

UIIO (input or input-output) and UOIO (output or input-output). Then during region searching events can be promoted from a U* set to one of the others. A region is found when no event is in a U* set and the entry/exit relation of regions is satisfied. Initial scenario is defined by a back- and a fore-pattern. Back-pattern is `UONC`, by default, or the one determined by option `-b back-set`. Fore-pattern is none, the default, or the one determined by `fore-pattern`. This is a sequence of option of the form `-(I | O | IO | NC | UONC | UINC | UIIO | UOIO) event-list`. For instance, to print all regions of transition system `xpto.ts` having $a+$ as the input interface, execute the command `ts-regions -ts xpto.ts -b UONC -I a+`.

ts-disjoint

TYPE: C program

SYNOPSIS: `ts-disjoint [-ts infile] ev1 ev2`

DESCRIPTION: Given a transition system (default standard input) and the input interfaces of two regions, prints, on standard output, all pairs of regions, their intersection and points out those with empty intersection.

sig-factor

TYPE: Bourne shell script.

SYNOPSIS: `sig-factor [ -name name ] I1 O1 I2 O2 sin sig`

DESCRIPTION: Generates the TSF description of a signal insertion factor, following theorem 5.6. Position of initial state in the partition is given by `sin`, which must have values 0 if $\in \langle O2, I1\rangle$, 1 if $\in \langle I1, O2\rangle$, 2 if $\in \langle O1, I2\rangle$ and 3 if $\in \langle I2, O2\rangle$. Signal to insert is named `sig`. The generated transition system is named `name`, if option `-name` is given, or "sig-#", where # is a random number, otherwise.

ch-factor

TYPE: Bourne shell script.

SYNOPSIS: `ch-factor [ -name name ] I1 O1 I2 O2 sin req gr`

DESCRIPTION: Generates the TSF description of a genex channel insertion factor. Position of initial state in the partition is given by `sin`, which must have values 0 if $\in \langle O2, I1\rangle$, 1 if $\in \langle I1, O2\rangle$, 2 if $\in \langle O1, I2\rangle$ and 3 if $\in \langle I2, O2\rangle$. Request and grant signals to insert are named `req` and `gr`, respectively. The generated transition system is named `name`, if option `-name` is given, or "ch-#", where # is a random number, otherwise.

sig-ren

TYPE: C program.

SYNOPSIS: `sig-ren [-ts infile] sig new`

DESCRIPTION: rename signal transitions `sig`$\{+,-\}$ as `new`$\{+,-\}$ of `infile` (default standard input). Result printed in standard output.

ts-comp

TYPE: C program.

SYNOPSIS: `ts-comp [-q] [-i infile-1] infile-2`

DESCRIPTION: Compares transition systems given by `infile-1` (default *standard input*) with `infile-2`. In current version only says if they are equal or not, and do not print differences. Option `-q`, meaning quiet mode, suppress any output, only returning exit status.

stg-genex

TYPE: Bourne shell script.

SYNOPSIS: `stg-genex [-n name] n k [channel-options]`

DESCRIPTION: Generates the STG description of a genex $n \times k$, named `name`, if option `-n` is used, or `gx-#`, where # is a random number, otherwise. The `channel-options` have the form `-c r g i`, for $c = 1, 2, \cdots, n$, and represents channel information for channel `c`, `r` being the name of the request signal, `g` the name of the grant signal and `i` the position of the token in the channel cycle, 0 for a token in place $\langle g-, r+ \rangle$, 1 in place $\langle r+, g+ \rangle$, 2 in place $\langle g+, r- \rangle$ and 3 in place $\langle r-, g- \rangle$. By default, signals for channel `i` are called `ri` and `gi` and `i` is assumed to be 0.

5.6.5 Tools Evolution

The TSF toolset was developed to support the work carried out during the thesis. In that domain we are more interested in accessing and evaluating intermediate results than in getting the final result. Thus, having a set of tools, each one implementing a specific task, is more useful than having a single tool, which goes directly from the conflict specification to the final implementation. However, from the end user point of view this can be cumbersome. But, using the set of tools, we can easily build a working interface, preferably a graphical, user-friendly one, where the tools are integrated.

When we defined the TSF file format we had one idea in mind: making everything explicitly. Thus,

there is an event section to declare events, a state section to declare states and a transition section to declare transitions. The ASTG file format used by *petrify* does not have a section to declare places, when describing signal transition graphs, or states, when describing state graphs. In the transition section, new symbols appear representing either places or states. (See, for instance, the state graph shown in figure 5.7, where the symbols starting with an x represent state names.) Explicitly making the state declaration, parsing is made more robust. However, in most descriptions the state section has the form of an enumeration of state names, with a common prefix. For instance, the 50 states of a description can be named $x01, x02, \cdots, x50$. The need to write these 50 state names is unpleasant. We have incorporated in our *to do* list the adaptation of the TSF file format, in order to allow the use of ranges in the state section.

5.7 Conclusions

In chapter 4 we have analyzed non-persistent specifications and we have proposed a methodology to make their synthesis based on the inclusion of genexes, special arbitration devices capable of implementing an exclusion relation of k out of n. In this chapter we have evolved in order to obtain the procedure, which systematically goes from specification to implementation. The different synthesis steps were defined, mathematically formalized and tools have been developed to implement them. Proofs of correctness of the different synthesis steps were presented, often in the form of proved theorems.

The synthesis procedure assumes the existence of a component library, where the generic genex $n \times k$ is available. We have explained how to build a genex $n \times 1$ from the well-known 2-input mutex, which corresponding to the genex 2×1. Then we have proposed an implementation for the genex $n \times k$, based on the genex $n \times 1$.

A set of tools were developed in the Linux operating system. Each tool was constructed to carry out a specific task, and falls down in one of two categories. In one category, tools were written as Bourne shell scripts and are used to generate auxiliary data, like for instance the signal transition graph description of a genex or the transition system description of a factor representing the insertion of a genex channel into a given specification. The other category was developed in the C programming language and is more concerned with the analysis and manipulation of transition systems. These tools

grow on top of a common framework, which includes an object model and a function library. Data for the examples presented along the thesis were obtained using the developed set of tools.

Chapter 6

Conclusions

This chapter presents conclusions and summarizes the main contributions of the work described in this thesis, namely, the definition of a methodology which enlarges the class of specifications synthesizable as speed-independent asynchronous circuits. The set of tools developed to accomplish the synthesis methodology is also summarized. Finally, we discuss possible directions for future research.

6.1 Speed Independent Circuits

Speed independent circuits appear to be a quite interesting class to work with in the asynchronous domain. Their unbounded gate delay assumption makes them immune to technological parameter variations. Circuits can be safely ported to newer and faster technologies. Circuits also work correctly if some environment parameters, like operating temperatures or power supply, deviate from typical values. Its "zero" wire delay assumption is not a severe restriction because it can be assumed for several circuit implementation technologies. Its input-output mode of operation allows for highly concurrent specifications. Changes on input signals are allowed without waiting for the circuit to be completely stable. Only causal relations on the specification have to be observed. Other appealing features, like high modularity and efficient formal verification methods, can also be mentioned.

A robust specification formalism, both at state and event levels, allied to the existence of systematic synthesis methodologies have led to the emerging of complete automated synthesis tools. SIS [73] and Petrify [22] are examples of them. However some circuits, like arbiters and synchronizers, are

excluded. They present signal transition conditional relations which violate the speed independent assumption. Non-persistences are only allowed in specifications if only input signals are involved. Non-commutativities between input signals are only allowed if they do not violate the complete state coding property.

6.2 Contributions

This work contributes to the synthesis of asynchronous circuits from state graph specifications in different ways. We provide a theoretical structure to model non-persistent conflicts in state graph specifications. This structure is used to drive a transformation process which both assigns conflicts to external arbitration devices and generates a state graph description suitable to feed an existing speed independent logic synthesis tool. A set of tools that aid in accomplishing the transformation process was also developed.

6.2.1 Conflict Model

A methodology to synthesize state graph specifications containing non-persistence conflicts was presented. Supporting the methodology is a non-persistence conflict model, based on the notion of an exclusion relation of k out of n. It represents an exclusion relation among n different concurrent regions of the circuit specification, in the sense that the system can be inside of no more than k of those n regions. Under this conflict model, non-persistence conflict states are caused by these exclusion relations and thus we manage them controlling access to the associated concurrent regions.

A partially analog arbitration device that can fairly implement an exclusion relation of k out of n was introduced and denominated genex. Access to n concurrent regions in an exclusion relation of k out of n, with $k < n$, can be controlled by using a genex $n \times k$. The genex 2×1 corresponds to the well known mutex, a device which can fairly implement the mutual exclusion relation. We have shown how to build the general genex $n \times k$ using genexes 2×1 as the building block.

We have also shown that a set of concurrent regions in an exclusion relation of k out of n includes, if $n > k + 1$, n exclusion relations of k out of $n - 1$. Thus, genex insertions are based on maximal sets of concurrent regions, which are called conflict points.

Finally, in order to use the same model to cover both symmetric and asymmetric non-persistences, we have introduced an extension to the common definition of region. The extension takes form with the notion of empty region, a region without states and with a set of events being simultaneously its input and output interfaces.

6.2.2 Transformation Process

We have presented a transformation process which, starting with a non-persistence conflict specification, assigns the conflicts to genexes and generates a state graph description suitable to feed an existing speed independent synthesis tool. It is driven by conflict points. Thus the transformation process must look for conflict points and control them by the insertion of genexes.

We have presented a mechanism to find out conflict points. Analyzing a conflict specification we can draw the non-persistence graph. Complete components in this graph are associated to conflict points and partially define them. More specifically, from a complete component we can extract the number and the input interfaces of the concurrent regions which represent a conflict point. We conclude the conflict point identification using a region search mechanism.

Once a conflict point, covering one or more conflict states, is identified, it is necessary to transform the original specification in order to put a genex controlling access to the concurrent regions associated with that conflict point. A process to carry out such transformation was developed. It unfolds into two phases. The first transformation phase deals with input signals involved in non-persistences. The input signals are “stretched” and split into two signals, one of them internal to the circuit. Non-persistences are transferred to the internal signals.

In a second phase, the original specification or the specification resulting from the first phase is transformed in order to accommodate interaction with the genex selected to control the conflict point. The transformation takes the simple form of a product of two state graphs and a number of transition systems. We have shown that this transformation results in a valid state graph, which preserves the original specification with exception of the conflict point, which is removed.

A set of tools have been developed to implement most of the steps of the transformation process. These tools were used to produce the data for the several examples presented along the thesis.

6.3 Future Research

Some directions for future research and developments can be devised based on the work presented in this thesis.

State graph specifications containing non-commutative conflicts have been analyzed in order to define a methodology for their implementation as speed-independent circuits. A promising approach, elicited the 2-channel scheduler as a good candidate to control the non-commutativities in a specification. It is a 2-input, 2-output arbitration device, which accepts requests at any time and guarantees grants under mutual exclusion and in arrival order. It was presented as a theoretical component and thus a possible research line corresponds to the design of a physical implementation.

We have referred to non-commutativities situations involving only two signals, thus suggesting the use of the 2-channel scheduler to control them. For higher order cases schedulers with more channels must eventually be used. Their characterization and implementation must be investigated. Finally, a methodology to systematically control all the non-commutativities existent in a specification must also be investigated.

As already mentioned, tools were developed as a collection of command line applications, each implementing a different topic of the transformation process. Access and analysis of all the intermediate results and test of different alternatives are possible in this way. However it can be little friendly to end users, because they must be aware of a multitude of command names. A user-friendly interface, eventually a graphical one, can be developed to keep things together. This interface must be carefully designed in order not to disallow the choice of possible different implementation alternatives. Prior to that we also recommend the consolidation of the TSF toolset. The development of some tools have caused the adjustment of the support object model used so far. However, prior developed tools were not rebuilt to use the new object model, since their functionality was not affected.

Bibliography

[1] André Arnold. *Finite Transition Systems*. Computer Science. Prentice Hall, Paris, 1994.

[2] Marek Bednarczyk and Andrzej Borzyskowski. Concurrent realizations of reactive systems. In *Electronic Notes in Theoretical Computer Science, volume 29*. Elsevier Science B. V., 1999. http://www.elsevier.nl/locate/entcs/volume29.html.

[3] Marek Bednarczyk and Andrzej Borzyskowski. General morphisms of petri nets. Technical report, Institute of Computer Science, Polish Academy of Science, 1999. An extended (and revised) abstract appeared in Proc ICALP'99, Springer LNCS, vol. 1644, pages 190–199, 1999.

[4] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[5] Peter A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, 1994.

[6] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[7] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[8] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.

[9] L. Bernardinello, G. de Michelis, K. Petruni, and S. Vigna. On synchronic structure of transition systems. Technical report, Universitat di Milano, 1994.

[10] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[11] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[12] S. M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[13] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[14] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 407–410. IEEE Computer Society Press, 1985.

[15] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. International Conf. Computer Design (ICCD)*, pages 220–223. IEEE Computer Society Press, 1987.

[16] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[17] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[18] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[19] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing petri nets from state-based models. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 164–171, 1995.

[20] Jordi Cortadella. Petrify: a tutorial for the designer of asynchronous circuits. Technical report, Universitat Politècnica de Catalunya, 1999.

[21] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[22] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[23] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.

[24] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.

[25] Jordi Cortadella, Alexandre Yakovlev, Luciano Lavagano, and Peter Vanbekbergen. Designing asynchronous circuits from behavioral specifications with internal conflicts. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 106–115, November 1994.

[26] A. Davis, B. Coates, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[27] Al Davis. Synthesizing asynchronous circuits: Practice and experience. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 104–150. Springer-Verlag, 1995.

[28] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[29] J. B. Dennis and S. S. Patil. Speed-independent asynchronous circuits. In *Proc. Hawaii International Conf. System Sciences*, pages 55–58, 1971.

[30] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivations of programs. *Communications of the ACM*, 18(8):453–457, 1977.

[31] David L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI*, pages 51–65. MIT Press, 1988.

[32] C. Duboc. Mixed products and asynchronous automata. *Theoretical Computer Science*, 48:183–199, 1986.

[33] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[34] Jo C. Ebergen, John Segers, and Igor Benko. Parallel program and asynchronous circuit design. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.

[35] Sonya Gary *et al*. Powerpc 603 m«microprocessor, a low-power design for portable applications. In *Proc. Compcon*, pages 307–315, 1994.

[36] S. Furber. Computing without clocks: Micropipelining the ARM processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[37] S. B. Furber. *ARM System Architecture*. Addison-Wesley, Reading, MA, 1996.

[38] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[39] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.

[40] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[41] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[42] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[43] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, C-31(12):1133–1141, December 1982.

[44] D. A. Huffman. The synthesis of sequential switching circuits. *IRE Transactions on Electronic Computers*, 257(3):161–190, 1954.

[45] D. A. Huffman. The synthesis of sequential switching circuits. *IRE Transactions on Electronic Computers*, 257(4):275–303, 1954.

[46] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of self-timed circuits. In Jørgen Staunstrup and Robin Sharp, editors, *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 275–287. Elsevier Science Publishers, 1992.

[47] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[48] Alex Kondratyev, Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexander Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347–362, February 1999.

[49] Alex Kondratyev, Michael Kishinevsky, Bill Lin, Peter Vanbekbergen, and Alex Yakovlev. Basic gate implementation of speed-independent circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 56–62, June 1994.

[50] Luciano Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, U.C. Berkeley, November 1992. Technical report UCB/ERL M92/140.

[51] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[52] Kuan-Jen Lin, Chi-Wen Kuo, and Chen-Shang Lin. Synthesis of hazard-free asynchronous circuits based on characteristic graph. *IEEE Transactions on Computers*, 46(11):1246–1263, November 1997.

[53] K. S. Low and A. Yakovlev. Token ring arbiters: An exercice in asynchronous logic design with petri nets. Technical report, University of Newcastle upon Tiny, 1994.

[54] Alain J. Martin. A synthesis method for self-timed VLSI circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 224–229, Rye Brook, NY, 1987. IEEE Computer Society Press.

[55] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[56] O. Melnokov, V. Sarvanov, R. Tyshkevich, V. Yemelichev, and I. Zverovich. *Exercices in Graph Theory*. Kluwer Texts in the Mathematical Sciences. Kluwer Academic Publishers, 1998.

[57] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[58] Cho W. Moon, Paul R. Stephan, and Robert K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 322–325. IEEE Computer Society Press, November 1991.

[59] R. Morin. Decomposition of asynchronous systems. In *Proc. CONCUR'98*, Lecture Notes in Computer Science, pages 549–564. Springer, 1998.

[60] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[61] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–574, April 1989.

[62] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theorical Computer Science*, 96:3–33, 1992.

[63] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[64] Steven M. Nowick and David L. Dill. Synthesis of asynchronous state machines using a local clock. In *Proc. International Conf. Computer Design (ICCD)*, pages 192–197. IEEE Computer Society Press, October 1991.

[65] E. Pastor, J. Cortadella, O. Roig, and A. Kondratyev. Structural methods for the synthesis of speed-independent circuits. In *Proc. European Design and Test Conference*, pages 340–347. IEEE Computer Society Press, March 1996.

[66] Enric Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs*. PhD thesis, Univsitat Politècnia de Catalunya, February 1996.

[67] Ad Peeters. The 'Asynchronous' Bibliography (BibTeX) database file `async.bib`. http://www.win.tue.nl/~wsinap/doc/async.bib. Corresponding e-mail address: `async-bib@win.tue.nl`.

[68] A. Pereira, J. Cortadella, and A. Ferrari. Synthesis of asynchronous circuits from petri net based specifications with conflicts. In *Fifth BELSIGN Workshop*, April 1997.

[69] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[70] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[71] Milton Sawasaki, Chantal Ykman-Couvreur, and Bill Lin. Externally hazard-free implementations of asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, June 1995.

[72] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[73] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[74] Polly Siegel and Giovanni De Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 558–565, November 1994.

[75] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[76] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[77] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 112–117. IEEE Computer Society Press, November 1992.

[78] Peter Vanbekbergen, Bill Lin, Gert Goossens, and Hugo de Man. A generalized state assignment theory for transformations on signal transition graphs. *Journal of VLSI Signal Processing*, 7(1/2):101–115, February 1994.

[79] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[80] J. V. Woods, P. Day, S. B. Furber, J. D. Garside, N. C. Paver, and S. Temple. AMULET1: An asynchronous ARM processor. *IEEE Transactions on Computers*, 46(4):385–398, April 1997.

[81] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 104–111. IEEE Computer Society Press, November 1992.

[82] Chantal Ykman-Couvreur and Bill Lin. Optimised state assignment for asynchronous circuit synthesis. In *Asynchronous Design Methodologies*, pages 118–127. IEEE Computer Society Press, May 1995.

[83] Chantal Ykman-Couvreur, Bill Lin, Gert Goossens, and Hugo De Man. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *Proc. European Conference on Design Automation (EDAC)*, pages 512–517. IEEE Computer Society Press, February 1993.

[84] Kenneth Y. Yun and David L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.

[85] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.